COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Cours numéro 1                         *
*                                                                *
******************************************************************
```

Ce cours d'assembleur pourra sembler réduit de par sa taille. Il ne l'est cependant pas par son contenu! L'assembleur est le langage le plus simple qui soit, pourvu qu'il soit expliqué simplement ce qui n'est malheureusement pas souvent le cas. C'est un peu le pari de ces cours: vous apprendre ce qu'est l'assembleur en une quinzaine, pas plus. De toutes façons, à part grossir la sauce avec du bla-bla inutile, je ne vois pas très bien comment faire pour que les cours durent plus de 15 jours. Evidement, vous comprenez bien que les cours payants ont eux, tout à fait intérêt à faire durer le plaisir et à vous faire croire que c'est très très dur à comprendre et qu'il va falloir au moins 2568478 leçons si vous voulez vous en sortir!!!

Ce cours est destiné au débutant, il est composé de plusieurs parties relativement courtes mais dans lesquelles

TOUT EST IMPORTANT.

PRESENTATION ET AVERTISSEMENT

Pour programmer en ASM, plusieurs habitudes sont nécessaires. Autant les prendre dès le début car, très vite, ce qui apparaissait comme de petits problèmes peut tourner à la catastrophe.

Tout d'abord avoir de l'ordre au niveau disquette: Les sauvegardes sont très nombreuses et c'est vite la pagaille. Travailler avec soins: commentaires clairs et précis sur les listings, éviter les soit disant 'trucs' qu'on ne comprendra plus au bout de 3 jours, mettez quelques lignes explicatives au début du listing...

Au niveau outil, je conseille DEVPAC pour sa facilité d'emploi, et sa doc qui, bien qu'en Anglais et suffisamment claire pour que l'on y retrouve facilement les informations.

Si vous avez un 1040 (ou plus), n'hésitez pas à demander au niveau des 'préférences' de GENST, un chargement automatique de MONST, le débugger.

Pour ce qui est des livres de chevet (et de travail), il faut bien sur faire la différence entre 2 types d'ouvrages: ceux relatifs au 68000 Motorola et ceux relatifs à l'ATARI. Il faut ABSOLUMENT un ouvrage de chaque. Attention, pour celui relatif au 68000, il ne doit pas s'agir d'un ouvrage de vulgarisation, mais d'un ouvrage technique, qui vous semblera certainement incompréhensible au

début.

Par exemple documentation du fabricant de microprocesseur (MOTOROLA ou THOMSON pour la France). Cherchez du côté des vendeurs de composants électroniques plutôt que dans les magasins de micro-ordinateurs. En désespoir orientez-vous vers "Mise en oeuvre du 68000" aux éditions SYBEX.

Une remarque qui devra IMPERATIVEMENT guider votre choix:

Le vocabulaire informatique est composé en très grande partie d'abréviations. Or ce sont des abréviations de termes anglais.

Il est donc impératif que l'ouvrage sur le 68000 donne la signification de chacune des abréviations, signification en Anglais et traduction en Français. Attention de très nombreux ouvrages ne donnent que la traduction or autant il est difficile de se souvenir de la signification de termes tels que DATCK, BG, BGACK, MMU ou MFP, autant leurs fonctions et clairs et surtout facilement mémorisable si on connaît la signification de ces abréviations dans la langue original, la traduction coulant de source!

Pour l'ouvrage relatif au ST, le must consiste à se procurer chez ATARI la DOCumentation officielle pour les Développeurs. Sinon, "la Bible" ou "le Livre du développeur" chez Micro Application, même s'il y subsiste quelques erreurs, est un excellent palliatif.

A part cela, n'achetez aucun autre ouvrage : "le livre du GEM", "Graphismes en ASM", "cours d'ASM" etc, ne seront que des gouffres pour votre porte-monnaie et ne vous apporteront rien.

Si, après ces achats il vous reste quelque argent, je ne peux que vous conseiller très fortement l'achat d'une calculatrice possédant les opérations logiques (AND, OR, XOR...) et les conversions entre bases utilisées en informatique (binaire, hexadécimal...).

Je vous conseille la Texas Instrument TI34. C'est une calculatrice 'solaire' (mais qui marche avec n'importe quelle source lumineuse), qui à l'avantage d'être simple à manipuler. Vous la trouverez à un prix variant de 199 Frs (magasin NASA) à 240 Frs (Camif ou la Fnac). C'est une dépense qui n'est pas trop élevée et qui s'avérera très vite rentable!

METHODE DE PENSÉE D'UN ORDINATEUR

En France nous roulons à droite. C'est simple, entré dans les moeurs, et tout le monde s'en sort bien. Imaginons la conduite en Angleterre... Pour un Français il existe en fait 3 solutions:

1) On ne lui dit rien sur ce type de conduite :
    C'est avantageux dans le sens ou notre conducteur part tout de suite sur la route, mais bien sûr le premier carrefour risque de lui être fatal.

2) On lui ré-apprend à conduire de A à Z :
    C'est long, on a l'impression de perdre son temps, mais on limite presque totalement les risques d'accidents.

3) On dit simplement au conducteur: Attention, ici on roule à gauche.
    Celui-ci, sait conduire à droite, en le prévenant il fera attention et s'en sortira. Avantage: c'est rapide, inconvénient: un simple relâchement et c'est l'accident.

Programmer, c'est comme vouloir conduire à gauche. Il suffit de penser, mais pas de penser comme nous, mais comme la machine. Conscient de votre volonté d'aller vite, c'est la méthode 3 que nous allons utiliser, mais attention au relâchement.

Un dernier conseil avant de vous laisser aborder le premier cours à proprement parler: l'assembleur plus que tout autre langage, et assimilable à une construction en Lego. Une énorme construction en Lego n'est pourtant qu'un assemblage de petites briques. Assembler 2 briques et passer 1 ou 2 heures pour étudier cet assemblage peut paraître inutile. Pourtant c'est ce que nous allons faire: il y a peu de choses à apprendre mais elles sont très importantes. On ne le répétera jamais assez: ce ne sera pas quand notre château de Lego d'un mètre cinquante commencera à s'écrouler qu'il faudra se dire "merde ,mes 2 petites briques du début étaient peut être mal fixées", car à ce moment-là, alors qu'une machine accepterait de tout reprendre dès le début, il y a 99% de chances pour que votre expérience en ASM s'arrête là, ce qui serait dommage!

De même, je vous déconseille fortement la chasse aux listings!

Cette pratique est très courante entre autre sur RTEL et n'amène généralement que des ennuis! Il est de TRES LOIN préférable de passer pour un con parce qu'on ne sait pas faire un scrolling plutôt que de frimer alors qu'on a juste recopié un source que nous a donné un copain! A ce petit jeu là, il y a des gagnants en basic, en C ou en Pascal mais jamais en assembleur, car lorsque vous commencerez à vouloir coller des sources entre eux et que ça ne marchera pas, vous serez TOTALEMENT incapable de comprendre pourquoi, et il sera trop tard pour apprendre et vous abandonnerez. Et ne dites pas non, regarder plutôt 6 mois en arrière sur RTEL et souvenez vous de ceux qui faisaient alors de l'ASM, ils ont presque tous abandonnés! N'oubliez pas non plus une différence fondamentale entre un langage quelqu'il soit et l'assembleur: Il faut environ 6 mois pour apprendre le C ou le Pascal. Ensuite le temps sera passé à produire de bons algorithmes, et à taper les programmes.

En assembleur il en est tout autrement. En un mois maximum le 68000 ne devrait plus avoir de secret pour vous, par contre tout le temps qui suivra devra être consacré à faire des recherches plus ou moins évidentes sur des 'trucs' à réaliser plus vite, avec plus de couleurs etc... Un programmeur en BASIC ou en C recherche des sources pour travailler. Pas un programmeur en assembleur! Le programmeur en assembleur VA FAIRE les routines! Typiquement on va demander à un programmeur en C de faire un programme et le programmeur en C va demander au programmeur en assembleur de réaliser la ou les routines soi-disant infaisables! Et bien sur pour ces routines, pas de sources de distribuées!!!! Ce que nous apprendrons donc ici, c'est à programmer comme des vrais! A chercher, à comprendre afin de pouvoir par la suite chercher tout seul.

Si vous vous attendez à trouver dans ce cours des sources entiers de scrolling, de lectures de digits ou de gestion de souris sans le GEM, vous faites fausse route! Retourner au basic que vous n'auriez jamais dû quitter; Vous resterez a tout jamais ce que l'on appelle un lamer dans les démos, celui qui recopie mais ne comprend rien.

Si par contre vous voulez savoir, alors accrochez vous car les infos sont rares mais ... quel plaisir lorsqu'après plusieurs nuits blanches vous verrez apparaître votre premier scrolling dont vous pourrez dire : "c'est moa qui l'ai fait!!!", et là ce sera vrai!!

Dans ce cours nous étudierons le 68000 mais également les particularités du ST: les interruptions par le MFP68901, le son (digit ou non), les manipulations graphiques, l'interface graphique Ligne A, et enfin un gros morceau, souvent critiqué mais toujours utilisé, le GEM.

Bon courage !

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Cours numéro 2                         *
*                                                                *
******************************************************************
```

LES CHIFFRES 'MAGIQUES'

Voyons d'abord d'une façon simple comment marche un ordinateur, en nous plaçant dans la situation suivante: nous devons fournir des messages à une personne dont nous sommes séparés (par exemple, message de nuit entre des gens éloignés).

Nous avons une lampe de poche, que nous pouvons donc allumer, ou éteindre, c'est tout. Nous pouvons donc donner 2 messages 1)la lampe est éteinte (par ex. tout va bien) 2)la lampe est allumée (par ex. vla les flics!)

Approfondissons les 2 états de la lampe:

	Allumée	Eteinte
qui revient à:	du courant	pas de courant
ou: Du courant ?	OUI	NON
Valeur du courant ?	1	0

Les tests seront donc notés par 0 ou 1 suivant l'allumage ou non de la lampe.

Comme nous sommes riches, nous achetons une 2 ème lampe.
Nous avons donc 4 possibilités de message

LAMPE 1	LAMPE2
éteinte	éteinte
allumée	éteinte
éteinte	allumée
allumée	allumée

En comptant avec 3,4,5,6 ... lampes, nous nous rendons compte qu'il est possible de trouver une relation simple entre le nombre de lampes et le nombre de possibilités.

Nombre de possibilités = 2 à la puissance nombre de lampes.

Nous obtenons donc le tableau suivant
Les remarques sont justes là pour mettre la puce à l'oreille !

Lampes	Possibilités	Remarques
1	2	
2	4	
3	8	Il y a des ordinateurs 8 bits ...
4	16	et des 16 bits...
5	32	Le ST est un 16/32 bits
6	64	Amstrad CPC... 64!!
7	128	ou Commodore 128 ?
8	256	En informatique le codage des caractères (lettres chiffres.. grâce au code ASCII) permet d'avoir 256 caractères !
9	512	Un 520 a 512 Ko de mémoire et Amstrad vend un PC1 512
10	1024	La taille mémoire de mon 1040!
11	2048	Celle du méga 2 de mon frère
12	4096	Celle d'un méga 4. Aussi le nbr de couleurs affichables avec un Amiga.
etc...		
16	65536	Dans le GFA, un tableau ne peut avoir plus de 65536 éléments.

Si mes 4 lampes sont éteintes (0000) je suis donc à la possibilité 0. Si elles sont allumées (1111) je suis donc à la 15 (car de 0 à 15 ça fait bien 16) donc 0000 --> 0 et 1111 --> 15

J'ai donc un bouquin de 16 pages donnant les possibilités des 16 allumages possibles, et mon correspondant a le même. Comment faire pour lui envoyer le message de la page 13 ?

Le chiffre le plus petit étant à droite (on note les chiffre dans l'ordre centaines, dizaines, unités), plaçons les lampes.

Lampe numéro: 4 3 2 1
a) je n'ai qu'une lampe (la 1) elle est allumée donc j'obtiens la valeur 1. (je ne peut obtenir que 0 ou 1)

b)j'ai 2 lampes (1 et 2), allumées toutes les deux, j'obtiens la 4ème possibilité . J'ai donc la valeur 3 (puisque je compte les valeurs 0,1,2 et 3, ce qui en fait bien 4) Puisque la lampe 1 vaut au maximum la valeur 1, j'en déduis que la lampe 2 vaut à elle seule au maximum la valeur 2.

En effet lampe 1 allumée --> valeur 1
Lampe 2 allumée --> valeur 2
Donc les 2 allumées ensemble --> valeur 3 = 4 possibilités.

La lampe 2 peut donc donner une 'augmentation' de 0 ou de 2.

Lampe numéro	4	3	2	1
'augmentation'	8	4	2	1

Pour envoyer le message 13, il faut donc allumer la lampe 4 (valeur de 8), la lampe 3 (valeur de 4) et la 1 (valeur de 1)

Lampe	4		3		2		1
Etat de la lampe	1		1		0		1
Valeur	8	+	4	+	0	+	1 = 13

Nous sommes donc en train de compter en binaire.

En décimal : déc signifie 10, car un chiffre peut prendre 10 valeurs (de 0 à 9).

En binaire :bi = deux car chaque chiffre ne peut prendre que 2 valeurs (0 ou 1).

L'informatique est un domaine Anglo-saxon. Un 'chiffre binaire', en Anglais, ça se dit 'binary digit'. On garde la première lettre et les 2 dernières, et on dit qu'un chiffe binaire c'est un BIT !!! Un bit peut donc être à 0 ou 1. C'est la plus petite unité informatique,
car, le correspondant à qui nous envoyons des messages, c'est en fait un ordinateur. Au lieu d'allumer des lampes, nous mettons du courant sur un fil ou non. Un ordinateur 8 bits à donc 8 fil sur lesquels on met ou non du courant !

Pour envoyer des messages nous allons donc préparer des lampes avec des petits interrupteurs puis, quand nos lampes seront prêtes, on actionnera l'interrupteur principal pour envoyer le courant et donc allumer d'un coup les lampes prévues.

Nous allons donc, par l'intermédiaire de nos 'lampes', envoyer des messages au coeur de la machine (dans le cas du ST c'est un microprocesseur 68000 de chez MOTOROLA) qui a été fabriqué pour répondre d'une certaine manière aux différents messages.

On prépare donc nos lampes puis on allume. Nous, nous avons 16 lampes. En effet le 68000 Motorola
est un micro-processeur 16 bits.

Voici donc un 'programme' (c'est-à-dire une succession d'ordres) tel qu'il est au niveau mise ou non de courant sur les 16 fils Tout à gauche c'est la valeur du fil 16 et à droite celle du 1. 0 = pas de courant sur le fil, 1 du courant. Le microprocesseur est entouré de multiples tiroirs (les cases mémoire) et parmi les ordres qu'il sait exécuter il y a 'va chercher ce qu'il y a dans tel tiroir' ou bien 'va mettre ça dans tel tiroir'. Chaque tiroir est repéré par une adresse (comme chaque maison), c'est-à-dire par un numéro.

Nous allons dire au microprocesseur: va chercher ce qu'il y a au numéro 24576, ajoutes-y ce qu'il y a au numéro 24578 et mets le résultat au numéro 24580. On pourrait remplacer 'au numéro' par 'à l'adresse'.

Allumons donc les 16 lampes en conséquences, cela donne:

```
0011000000111000
0110000000000000
1101000001111000
0110000000000010
0011000111000000
0110000000000100
```

Une seule biarque s'impose, c'est la merde totale! Comment faire pour s'y retrouver avec un programme comme ça, si on oublie d'allumer une seule lampe, ça ne marche plus, et pour repérer l'erreur dans un listing pareil, bonjour !
la merde !!!!

On a donc la possibilité de marquer ça non pas en binaire, mais en

décimal. Malheureusement la conversion n'est pas commode et de toute façon, on obtient quand même des grands chiffres (visuellement car leur taille en tant que nombre ne change pas, bien sûr!) Ainsi la 3ème ligne donne 53368. On va donc convertir autrement, en séparant notre chiffres binaire en groupe de 4 bits.

REMARQUE DE VOCABULAIRE:

Nous ne parlerons qu'Anglais. Toutes les abréviations en informatique sont des abréviations de mots ou d'expressions anglaises. Les lire à la Française impose d'apprendre par coeur leur signification. En les lisant telles qu'elles DOIVENT être lues (en Anglais), ces expressions donnent d'elles mêmes leur définition. Un des exemples est T$ qui est lu systématiquement T dollar ! Or, $ n'est pas, dans le cas présent, l'abréviation de dollar mais celle de string. T$ doit donc se lire ET SE DIRE T string. String signifiant 'chaîne' en Anglais, T est donc une chaîne de caractère. Evident, alors que lire T dollar ne signifie absolument rien ! Le seul intérêt c'est que ça fait marrer Douglas, le joyeux britannique qui programme avec moi!

Une unité binaire se dit donc BIT (binary digit)
4 unités forment un NIBBLE

8 unités forment un octet (que nous appellerons par son nom anglais c'est à dire BYTE)

16 unités forment un mot (WORD)

32 unités forment un mot long (LONG WORD)

Revenons donc à notre conversion en groupant nos 16 lampes (donc notre WORD) en groupes de 4 (donc en NIBBLE)

```
 0011         0000          0011          1000
```

Ces 4 nibbles forment notre premier word.

Comptons dons les valeur possibles pour un seul nibble.

```
état du nibble 0000         valeur 0
        0001         valeur 1
        0010         valeur 2
        0011         valeur 3
        0100         valeur 4
        0101         valeur 5
        etc..
        1010         valeur 10
```

STOP ça va plus ! 10 c'est 1 et 0 or on les a déjà utilisés!
Ben oui mais à part 0,1,2,3,4,5,6,7,8,9 on n'a pas grand chose à notre disposition... Ben si, y'a l'alphabet !
On va donc écrire 10 avec A, 11 avec B, 12 avec C, 13/D, 14/E et 15 avec F. Il y a donc 16 chiffres dans notre nouveau système (de 0 à F). 'Déc' signifiant 10 et 'Hex' signifiant 6 (un hexagone) donc Hex + Déc=16. Décimal = qui a 10 chiffres (0 à 9) hexadécimal= qui en à 16!!!

Notre programme devient donc en hexadécimal:
$3038
$6000
$D078

$6002
$31C0
$6004

Plus clair mais c'est pas encore ça.

NOTE: pour différencier un nombre binaire d'un nombre décimal ou d'un hexadécimal, par convention un nombre binaire sera précédé de %, un nombre hexadécimal de $ et il n'y aura rien devant un nombre décimal. $11 ne vaut donc pas 11 en décimal, mais 17.

Réfléchissons un peu. Nous avons en fait écrit:

'Va chercher ce qu'il y a'
'à l'adresse $6000'
'ajoute y ce qu'il y a' 'à l'adresse $6002'
'met le résultat'
'à l'adresse $6004'

Le microprocesseur peut bien sûr piocher dans les milliers de cases mémoire qu'il y a dans la machine, mais en plus il en a sur lui (des petites poches en quelque sorte, dans lesquelles il stocke temporairement des 'trucs' dont il aura besoin rapidement). Il a 17 poches: 8 dans lesquelles il peut mettre des données, et 9 dans lesquelles il peut mettre des adresses. Donnée =DATA et adresse=ADRESS, ces poches seront donc repérées par D0,D1,D2, ...D7 et par A0,A1...A7 et A7' (nous verrons plus tard pourquoi c'est pas A8, et les différences entre ces types de poches).

NOTE: le phénomène de courant/pas courant et le même pour TOUS les ordinateurs actuels. Le nombre de 'poche' est propre au 68000 MOTOROLA .
Il y a donc le même nombre de 'poches' sur un Amiga ou un Mac Intosh puisqu'ils ont eux aussi un 68000 Motorala. Sur un PC ou un CPC, les caractéristiques (nombre de lampes allumables simultanément, nombre de 'poches'...)sont différents, mais le principe est le même. C'est allumé OU c'est éteint.

Modifions notre 'texte', qui devient donc.

'déplace dans ta poche D0'
'ce que tu trouveras à l'adresse $6000'
'ajoute à ce que tu as dans ta poche D0'
'ce que tu trouveras à l'adresse $6002'
'mets le résultat de l'opération'
'à l'adresse $6004'

La machine est très limitée, puisque par conception, le résultat de l'opération de la 3 ème ligne ira lui même dans D0, écrasant donc ce qui s'y trouve. Pour garder la valeur qui s'y trouvait il faudrait au préalable la recopier par exemple dans la poche D1!!!

Déplacer se dit en Anglais MOVE
Ajoute   se dit en Anglais ADD

Notre programme devient donc

```
MOVE    ce qu'il y a en $6000   dans  D0
ADD     ce qu'il y a en $6002     à   D0
MOVE    ce qu'il y a maintenant dans  D0 à $6004
```

C'est à dire:

```
MOVE    $6000,D0
```

```
ADD     $6002,D0
MOVE    D0,$6004
```

Nous venons d'écrire en clair un programme en langage machine.

La différence fondamentale avec un programme dans n'importe quel autre langage, c'est que là, chaque ligne ne correspond qu'à UNE SEULE opération du microprocesseur, alors que PRINT "BONJOUR" va lui en faire faire beaucoup. Il est évident que notre BASIC n'étant qu'un traducteur 'mécanique' sa traduction a toutes les chances d'être approximative, et, bien qu'elle soit efficace, elle utilise beaucoup plus d'instructions (pour le microprocesseur) qu'il n'en faut réellement.

Il faut bien aussi avoir une pensée émue pour les premiers programmeurs du 68000 qui ont d'abord fait un programme avec des 1 et des 0, programme qui ne faisait que traduire des chiffres hexadécimaux en binaires avant de les transmettre à la machine. Il ont ensuite réalisé, en hexadécimal des programmes traduisant des instructions du genre MOVE, ADD etc... en binaire...

Il suffisait ensuite de regrouper plusieurs instructions de ce type sous une autre appellation (incomprise directement par la machine) et de faire les traducteurs correspondants, et créer ainsi les langages 'évolués' (PASCAL, C, BASIC ...)

Nous allons donc nous intéresser à la programmation ou plutôt à la transmission d'ordre au 68000 Motorola. Combien d'ordres peut-il exécuter. Uniquement 56 !!!! (avec des variantes quand même mais ça fait pas beaucoup). Des recherches (à un niveau bien trop haut pour nous!) on en effet montrées qu'il était plus rapide d'avoir peu d'instructions faisant peu de chose chacune et donc exécutables rapidement les unes après les autres, plutôt que d'avoir beaucoup d'instructions (le microprocesseur perdant sans doute du temps à chercher celle qu'on lui a demandé de faire) ou bien des instructions complexes.

Travail à faire: relire tout ça au moins 2 fois puis se reposer l'esprit avant de lire la suite.

CONSEIL: ne commencez pas la suite tout de suite.
Avalez parfaitement TOUT ce qui est marqué, car la compréhension du moindre détail vous servira.

Une lampe, ce n'est pas grand chose, mais une de grillée et vous comprendrez la merde que ça amène.
Là, c'est pareil. La plus petite chose incomprise et vous n'allez rien comprendre à la suite. Par contre si tout est compris, la suite sera aussi facile, et surtout aussi logique.

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
```

```
*                                                                 *
*                  par Le Féroce Lapin (from 44E)                 *
*                                                                 *
*                          Cours numéro 3                         *
*                                                                 *
*******************************************************************
```

Si vous avez correctement étudié les deux premières leçons, vous devez normalement avoir un peu plus d'ordre qu'au départ, et le binaire et l'hexadécimal ne doivent plus avoir de secret pour vous.

Avant de commencer je dois vous rappeler quelque chose d'essentiel: Il est tentant de réfléchir en chiffre alors que bien souvent il serait préférable de se souvenir qu'un chiffre n'est qu'une suite de bits. Ainsi imaginons un jeu dans lequel vous devez coder des données relatives à des personnages. En lisant ces données vous saurez de quel personnage il s'agit, et combien il lui reste de point de vie. Admettons qu'il y ait 4 personnages. Combien faut-il de bits pour compter de 0 à 3 (c'est-à-dire pour avoir 4 possibilités) seulement 2 bits. Mes personnages peuvent avoir, au maximum, 63 points de vie (de 0 à 63 car à 0 ils sont morts), il me faut donc 6 bits pour coder cette vitalité. Je peux donc avoir sur un seul byte (octet) 2 choses totalement différentes: avec les bits 0 et 1 (le bit de droite c'est le bit 0, le bit le plus à gauche pour un byte est donc le 7) je code le type de mon personnage, et avec les bits 2 à 7 sa vitalité.

Ainsi le chiffre 210 en lui même ne veut rien dire. C'est le fait de le mettre en binaire: 11010010 et de penser en regroupement de bits qui va le rendre plus clair. Séparons les 2 bits de droite: 10 ce qui fait 3 en décimal, je suis donc en présence d'un personnage de type 3.

Prélevons maintenant les 6 bits de gauche: 110100 et convertissons.

Nous obtenons 52. Nous sommes donc en présence d'un personnage de type 3, avec 52 points de vitalité.

Ceci devant maintenant être clair, passons à une explication succinte concernant la mémoire, avant d'aborder notre premier programme.

STRUCTURE DE LA MEMOIRE

La mémoire, c'est un tube, très fin et très long. Il nous faut distinguer 2 choses:

1) Ce qu'il y a dans le tube.
2) La distance par rapport au début du tube.

ATTENTION, cette notion doit être parfaitement comprise car elle est perpétuellement source d'erreur.
Grâce à la distance, nous pourrons retrouver facilement ce que nous avons mis dans le tube. Cette distance est appelé 'adresse'.

Le tube a un diamètre de 1 byte (octet). Lorsque je vais parler de l'adresse $6F00 (28416 en décimal), c'est un emplacement. A cet emplacement je peux mettre un byte. Si la donnée que je veux mettre tiens sur un word (donc 2 bytes car 1 word c'est bien 2 bytes accolés), cette donnée occupera l'adresse $6F00 et l'adresse $6F01.

Imaginons que je charge une image (32000 octets) à partir de l'adresse $12C52. Je vais donc boucler 32000 fois pour déposer mon image, en augmentant à chaque fois mon adresse.

Imaginons maintenant que je veuille noter cette adresse. Je vais par exemple la noter à l'adresse $6F00.

Donc si je me promène le long du tube jusqu'à l'adresse $6F00 et que je regarde à ce niveau là dans le tube, j'y vois le chiffre $12C52 codé sur un long mot (les adresses sont codés sur des longs mots). Ce chiffre occupe donc 4 emplacements de tube correspondant à $6F00, $6F01, $6F02 ,$6F03. Or, que représente ce chiffre: Une adresse, celle de mon image!!!! J'espère que c'est bien clair...

Un programme, c'est donc pour le 68000 une suite de lectures du contenu du tube. Il va y trouver des chiffres qu'il va interpréter comme des ordres (revoir le cours 2). Grâce à ces ordres, nous allons lui dire par exemple de continuer la lecture à un autre endroit de ce tube, de revenir en arrière, de prélever le contenu du tube et d'aller le déposer autre part (toujours dans ce même tube bien sûr) etc... Pour savoir à quel endroit le 68000 est en train de lire les ordres qu'il exécute, il y a un compteur. Comme ce compteur sert pour le programme, il est appelé Program Counter, en abrégé PC.

Le 68000 a un PC sur 24 bits, c'est-à-dire qu'il peut prendre des valeurs comprises entre 0 et 16777215. Comme chaque valeur du PC correspond à une adresse et qu'en face de cette adresse (donc dans le tube) on ne peut mettre qu'un octet, une machine équipée d'un 68000 peut donc travailler avec 16777215 octets, ce qui fait 16 Méga. A titre indicatif, le 80286 de chez Intel qui équipe les 'gros' compatibles PC, ne comporte qu'un PC sur 20 bits ce qui restreint son espace à 1 méga...

A noter que la mémoire est destinée à recevoir des octets mais que ce que représente ces octets (texte, programme, image...) n'a strictement aucune importance.

PREMIER PROGRAMME

Nous allons tout de suite illustrer notre propos. Nous lançons donc GENST. Ceux qui ont un écran couleur devront le lancer en moyenne résolution, c'est préférable pour un meilleur confort de travail.

Même si vous avez un 520, choisissez dans les 'préférences' de GENST (dans le menu 'Options') un chargement automatique de MONST (Load MONST 'YES') mettez un Tab Setting de 11 et auto-indent sur YES.

Si MONST est déjà chargé son option dans le menu 'program' doit être disponible, sinon elle est en gris. Si c'est le cas, après avoir sauvé les préférences, quitter GENST et relancez le.

Maintenant, nous allons réaliser le programme suivant:

```
    Met le chiffre $12345678 dans le registre D0
    Met le chiffre $00001012 dans le registre D1
    Additionne le registre D0 avec le registre D1
```

Tout d'abord il faut savoir que ces ordres seront mis dans le tube, et qu'il nous faudra parfois repérer ces endroits. Pour cela nous utiliserons des étiquettes, que nous poserons à côté du tube.

Ces étiquettes (ou Label en Anglais) sont à inscrire tout à gauche dans notre listing alors que les instructions (ce qui est à mettre DANS le tube) seront inscrites après un espace ou mieux pour la lisibilité, après une tabulation.

Ainsi notre programme devient:

```
       MOVE.L     #$12345678,D0
       MOVE.L     #$00001012,D1
       ADD.L      D0,D1
```

Remarquer le signe # avant les chiffres. Le signe $ indique que ces chiffres sont inscrits en hexadécimal. Le signe # indique que c'est la valeur $12345678 que nous voulons mettre dans D0.

Si nous avions fait MOVE.L $12345678,D0, c'est la valeur se trouvant à l'adresse $12345678 que nous aurions mis en D0.

Pourquoi y a t-il .L après les MOVE et le ADD ? Nous verrons cela dans quelques minutes.

Pour le moment assemblons en maintenant appuyé [ALTERNATE] puis en appuyant sur A.

Normalement, tout s'est bien passé ou alors c'est que vous n'avez pas scrupuleusement recopié ce 'programme'.

Maintenant, débuggons notre programme, en maintenant appuyé [ALTERNATE] et en appuyant sur D.

Hop, nous nous retrouvons dans MONST qui, étant appelé à partir de GENST, a automatiquement chargé notre programme.

Jetons tout d'abord un coup d'oeil à ce ramassis de chiffre...

En haut nous retrouvons nos registres de données D0 à D7 ainsi que nos registres d'adresses A0 à A7 avec en prime A7'. Sous les registres de données, nous voyons SR et en dessous PC. Nous pouvons remarquer que PC nous montre une adresse et la première ligne de notre programme. Le PC indique donc ce qui va être exécuté.

La fenêtre du dessous (numéro 2) montre notre programme. Sur la gauche de cette fenêtre nous voyons les adresses. Symboliquement nous pouvons dire que la partie droite de cette fenêtre montre nos instructions dans le tube et que les chiffres de gauche nous indique l'endroit, l'adresse par rapport au début du tube.

La fenêtre de droite (la 3) donne en fait la même chose que la 2, mais avec la vision du 68000. Nous avions vu dans le cours 2 que pour la machine notre suite d'ordres n'était qu'une suite de chiffres.

Lorsque nous avons assemblé, l'assembleur a simplement converti ligne par ligne notre programme en chiffres.

Normalement dans la fenêtre 2 vous devez voir notre programme avec en face de la première instruction, une petite flèche. Regardez l'adresse de cette instruction (c'est-à-dire le chiffre de gauche, qui indique a quel endroit dans le tube se trouve cet ordre). Avec un 1040 sous TOS 1.4, cela tourne autour de $61BF0.

NOTE: Le 68000 permet à un programme de se placer n'importe où.
Sur certains micro-processeurs les programmes doivent impérativement tous se placer au même endroit. Pour nous ce n'est pas le

cas, ce qui explique que si mon programme est en $61BF0 il n'en est pas forcement de même pour vous: c'est normal.

Regardez maintenant la fenêtre 3 et cherchez-y la même adresse que celle que vous avez lue dans la fenêtre 2 en face de notre première ligne de programme. Normalement si vous n'avez touché à rien cette adresse doit normalement être la première.

Vous devez y voir 203C12345678. C'est ainsi que le micro-processeur reçoit MOVE.L #$12345678,D0!!!

Retournons sur la fenêtre 2. Notons l'adresse de la seconde ligne de notre programme et soustrayons ce chiffre à l'adresse de la première ligne. Nous obtenons 6. Nous en déduisons donc que :

MOVE.L   #$12345678,D0 occupe 6 octets en mémoire.

Faisons maintenant avancer notre programme. Pour cela maintenez enfoncé [CONTROL] et appuyez une fois sur Z. La petite flèche a sauté sur la seconde ligne, cette même ligne est maintenant indiquée par le PC et notre registre D0 contient maintenant la valeur $12345678. MONST indique tous les chiffres en hexadécimal, vous commencez à comprendre l'intérêt de la calculatrice...

Continuons en refaisant Control+Z. C'est maintenant la ligne 3 de notre programme qui est indiquée par le PC tandis que D1 s'est trouvé rempli par $00001012.

Continuons avec Control+Z. L'addition entre D0 et D1 s'est effectuée. Comme nous l'avions vu dans le cours 2, les possibilités sont minimes car le résultat a écrasé l'ancienne valeur de D1. Pour réaliser D0+D1=D2 il aurait d'abord fallu transférer D1 dans D2 puis faire ADD.L D0,D2.

Dans notre cas, D1 contient maintenant la valeur $1234668A.

Notre programme n'ayant pas véritablement de fin, quittons le artificiellement en tapant Control+C.

SECOND PROGRAMME

Effacer le premier programme (alternate C) et tapez le suivant:

```
      MOVE.L     #$12345678,D0
      MOVE.W     D0,D1
      MOVE.B     D1,D2
```

Nous avons vu dans Monst que D0-D7 étaient des registres assez grands. Nous avons réussi à mettre $12345678 dans D0, ce qui donne quand même 305419896 en décimal! En effet le 68000 est un microprocesseur 16/32 bits ce qui fait que ces registres ne sont pas codés sur 16 bits mais sur 32.

32 bits, cela fait un long mot (Long Word). Dans notre premier programme, nous voulions que l'instruction MOVE agisse sur tout le registre donc sur un long mot, c'est pour cela que nous avions précisé .L après le move.

NOTE: Le vocabulaire est très important et demande un petit effort au début. Ainsi MOVE.L ne veut rien dire. Il convient de lire ce mnémonique (c'est ainsi que sont appelé les instructions assembleurs) MOVE LONG. D'ailleurs l'appellation mnémonique (qui a

rapport avec la mémoire, qui sert à aider la mémoire) est à rapprocher de mnémotechnique (capable d'aider la mémoire par des moyens d'association mentale qui facilitent l'acquisition et la restitution des souvenirs /CF dictionnaire Le Robert). Autant donc lire les instructions en Anglais ce qui facilitera grandement la compréhension.

Puisque notre registre D0 (comme les autres d'ailleurs) et codé sur un long mot, il contient donc 2 words côte-à-côte. Pour les distinguer nous appellerons celui de gauche word de poids fort et celui de droite word de poids faible. Chacun de ces words est lui même composé de 2 bytes, celui de gauche étant de poids fort et celui de droite de poids faible. De poids faible car les changement qu'il peut apporter à la totalité du nombre sont faible alors que les données de gauche (donc de poids fort) y apportent des variations importantes.

Assemblons notre programme et débuggons.

Exécutons la première ligne. Le résultat est le même que pour le premier programme: le PC indique la seconde ligne, tandis que D0 à reçu la valeur $12345678.

Maintenant exécutons la seconde ligne. Que dit-elle ?
MOVE.W   D0,D1

C'est-à-dire déplacer le contenu de D0 pour le mettre dans D1. Mais attention, le déplacement doit se faire sur un word (précisé par .W après le move. Cela se lit MOVE WORD). Or les opérations se font toujours sur le poids faible. Le MOVE va donc prélever le word de poids faible de D0 pour le mettre dans le word de poids faible de D1. Celui-ci va donc recevoir $5678.

Continuons en exécutant la troisième ligne. Celle-ci demande:
MOVE.B   D1,D2   (move byte d1 d2)

Donc transfert du byte de poids faible de D1 vers le byte de poids faible de D2. Regarder bien les registres et les valeurs qu'ils reçoivent!

Quittez maintenant le programme avec CONTROL+C

TROISIEME PROGRAMME

```
        MOVE.L     #$12345678,D0
        MOVE.L     #$AAAAAAAA,D1
        MOVE.W     D0,D1
        SWAP       D0
        MOVE.W     D0,D2
```

On efface le programme précédent, on tape celui-ci, on assemble puis on débugue. L'exécution de la première et de la seconde ligne ne doivent plus poser de problème.

Nous devons obtenir
D0=12345678
D1=AAAAAAAA

Exécutons maintenant la troisième ligne. Il y a bien transfert du word de poids faible de D0 vers le word de poids faible de D1. Nous constatons que le word de poids fort de D1 N'EST PAS AFFECTE

par ce transfert, et qu'il reste tout à fait indépendant du word de poids faible.

4ème ligne. Ce mnémonique 'SWAP' (To swap= échanger) va échanger les 16 bits de poids faible avec les 16 bits de poids fort. D0 va donc devenir 56781234.

Dernière ligne. Transfert du word de poids faible de D0 (qui maintenant est 1234 et plus 5678) vers le word de poids faible de D2.

Nous avons vu que D0 contenait en fait 2 données et que ces données étaient totalement indépendantes. Ceci permet une grande souplesse de travail mais demande aussi une grande rigueur car si au lieu de faire MOVE.W D0,D1 j'avais juste commis une faute de frappe en tapant MOVE.L D0,D1 j'écrasais le word de poids fort de D1 et après je me serais étonné de trouver 1234 dans D1 à l'endroit où je devrais encore trouver AAAA.

Nous voyons tout de suite les énormes avantages de ce système. Nous n'avons à notre disposition que 8 'poches' de données (D0 à D7) mais si nous ne voulons garder que des words, nous pouvons en mettre 2 par poche, c'est-à-dire 16 en tout. De même si notre codage ne se fait que sur des bytes, c'est 32 bytes que nous pouvons garder (4 par poche). Cela peut paraître assez évident mais par exemple sur l'Archimède, ce n'est pas possible. Sur cette machine, un registre contient un long word ou rien!

RESUMÉ DE VOCABULAIRE

MOVE.L = move long
MOVE.W = move word
MOVE.B = move byte

CONSEILS
Prenez votre temps, relisez tranquillement ce cours et les précédents. Vous voyez ce n'est pas bien dur l'assembleur!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
*                                                                *
*                  par Le Féroce Lapin (from 44E)                *
*                                                                *
*                         Cours numéro 4                         *
*                                                                *
******************************************************************
```

Nous allons aborder maintenant les registres d'adresse. Tout comme les registres de données, ces registres sont codés sur 32 bits (un long mot). Donc à priori aucune différence, puisque le micro-processeur ne connaît que des chiffres, que ceux-ci représentent des données ou des adresses, peu lui importe. C'est vrai en grande

partie et d'ailleurs sur certains micro-processeurs, il n'y a qu'un ou deux registres, qui peuvent contenir indifféremment adresse ou données.

Voyons, grâce à un exemple, les différences en ce qui concerne le 68000 MOTOROLA.

Tapons donc le programme suivant, après avoir, bien sûr, effacé l'ancien, et assemblons.

```
          MOVE.L     #$12345678,D0
          MOVE.B     #$AA,D0
          MOVE.L     #$12345678,A0
          MOVE.B     #$AA,A0
          MOVE.L     #$12345678,A1
          MOVE.B     A1,D1
```

L'assembleur note 2 erreurs et nous les annonce par 'invalid size at line 4' et la même chose pour 'line 6'. Puisque c'est la taille et non l'opération elle-même qui semble poser problème, nous en déduisons que le MOVE vers ou à partir d'un registre d'adresse, n'est pas possible sur un byte. Rectifions donc la ligne 4 et la ligne 6 en remplaçant les MOVE.B par des MOVE.W et ré-assemblons.

Note: Lorsque l'assembleur note une erreur, il donne la ligne où se situe celle-ci. Dans cette numérotation les lignes vides sont comptées.

Ainsi si vous aviez passé une ligne après MOVE.L #$12345678,D0 les erreurs auraient été annoncées ligne 5 et 7.

Cela fait déjà une différence puisque si vous regardez bien le programme, nous voulions réaliser une opération avec D0: Le remplir au maximum de sa taille, puis vérifier que le MOVE de la ligne 2, n'affecterait que le byte de poids faible, puis réaliser la même opération sur A0.

Impossible à priori. Tant pis, suite à notre modification, l'opération se déroulera donc sur un word au lieu d'un byte.

Débuggons notre programme. Première constatation: l'assembleur, voyant que les opérations ont lieu avec des registres d'adresse et non pas des registres de données, a automatiquement modifié les MOVE vers A0 et A1, pour les transformer en MOVEA, ce qui se lit MOVE ADDRESS

Exécutons le programme pas-à-pas. D0 prend la valeur $12345678, puis seul son byte de poids faible est modifié, D0 prenant alors la valeur $123456AA. Ensuite A0 prend la valeur $12345678. Après la ligne suivante, l'opération affectant le word, nous devrions avoir $123400AA. Et bien pas du tout! Nous obtenons $000000AA.

Nous venons donc de voir qu'un registre d'adresse est totalement influencé (donc sur un long mot) lorsqu'il est la destination de l'opération. Qu'en est-il donc lorsqu'il en est la source ?

Continuons donc notre programme, avec le remplissage de A1 et de D1. Nous constatons par la suite que seul le word de poids faible de A1 vient écraser celui de D1.

NOTE: $AA est bien en chiffre en hexadécimal. Si vous pensiez qu'il s'agissait de simples lettres de l'alphabet, dormez 1 ou 2 jours, et reprenez le cours à la première leçon!

De tout ceci nous déduisons 2 définitions:

REGISTRES DE DONNÉES: Chaque registre de données a une longueur de 32 bits. Les opérandes sous forme d'octet occupent les 8 bits de poids faible, les opérandes sous forme de mot, les 16 bits de poids faible et les opérandes longs, la totalité des 32 bits.

Le bit de poids le plus faible est adressé comme bit 0, le bit de poids le plus fort est adressé comme bit 31.

Lorsqu'un registre de données est utilisé soit comme opérande source, soit comme opérande destination, seule la partie appropriée de poids faible est changée. La partie restante de poids fort n'est ni utilisée, ni modifiée.

REGISTRES D'ADRESSE: Chaque registre a une longueur de 32 bits, et contient une adresse sur 32 bits. Les registres d'adresse n'acceptent pas une opérande dont la taille est l'octet. Par conséquent lorsqu'un registre d'adresse est utilisé comme opérande source, soit le mot de poids faible, soit l'opérande long dans sa totalité est utilisé, en fonction de la taille de l'opération.

Lorsqu'un registre d'adresse est utilisé comme destination d'opérande le registre entier est concerné, indépendamment de la taille de l'opération. Si l'opération porte sur un mot, tous les autres opérandes subissent une extension de signe sur 32 bits, avant que l'opération ne soit effectuée.

Définitions extraites du document réf EF68000 (circuit intégrés MOS THOMSON EFCIS), 45 avenue de l'Europe 78140 Velizy.

Dans ces définitions, nous remarquons un nouveau terme: opérande. C'est le terme qui désigne la valeur utilisée dans l'opération. Ainsi dans MOVE.W D0,D1 l'opérande source, c'est le word de poids faible de D0 alors que l'opérande destination, c'est le word de poids faible de D1.

Nous savons maintenant ce qu'est le PC, un registre de données, un registre d'adresse, nous avons un peu idée de ce que nous montre les fenêtre de MONST, continuons donc à décortiquer ce fabuleux outil !

Pour observer la fenêtre de MONST, si vous n'avez pas assemblé de programme, impossible d'utiliser Alternate+D. Il vous sera répondu qu'il n'y a pas de programme en mémoire. Tapez donc Alternate+M, vous voyez MONST apparaître, mais vous demandant quel fichier charger. Tapez ESC et nous voici tranquille pour une observation.

Nous voyons bien dans la fenêtre du haut nos registres de données et à droite nos registres d'adresse. Sous les registres de données SR puis PC. Le PC (program counter), nous savons ce que c'est, mais le SR ?

LE STATUS REGISTER

Le SR (prononcer Status Register, ce qui veut dire en Français registre d'état), est un registre codé sur un word (16 bits) et qui, comme son nom l'indique, nous renseigne sur l'état du micro-processeur.

Il est l'exemple frappant de ce que nous avons vu dans l'introduc-

tion du cours 3, à savoir qu'il est bien dangereux de traiter un ensemble de bits comme un simple chiffre, plus ou moins grand. Voyons la décomposition du Status Register.

```
numéro des bits 15--------------------------------0
appellation       T . S . . . I2 I1 I0 . . . X N Z V C
```

Tout d'abord il faut savoir que certains bits du SR ne sont pas utilisés. Ils sont ici symbolisés par un point chacun.

Commençons par la description des bits de droite, en commençant par le 0.

Le bit C (C signifie Carry donc retenue en Français).
Ce bit est mis à 1 lorsqu'il y a une retenue dans le bit le plus élevé (donc de poids le plus fort) de l'opérande objet, dans une opération arithmétique.

Le bit V (V signifie oVerflow donc dépassement en Français).
Imaginons une addition de 2 nombres positifs, lorsque le résultat va déborder les limites du registres, on obtiendra en fait un nombre négatif à complémente à 2. En effet le fait de mettre le bit de poids le plus fort à 1 indique que le nombre est négatif. Comme ce n'est pas,dans le cas présent, le résultat recherché, on est prévenu du dépassement par le fait que le bit V est mis à 1. Il indique également, lors de divisions, que le quotient est plus grand qu'un word ou bien que nous avons un dividende trop grand.

Le bit Z (Z signifie Zéro). Il n'indique pas que le résultat est égal à 0, mais plutôt que le résultat est passé de l'autre coté de 0. En effet, ce bit est à 1 lorsqu'après une opération le bit de poids le plus fort du résultat est mis à 1, ce qui signifie que nous sommes en présence d'un nombre négatif en complément à 2. Le bit N (N signifie Negate) signifie que nous sommes en présence d'un nombre négatif.

Le bit X (X signifie eXtend donc extension). C'est un bit bien spécial qui se comporte un peu comme une retenue. Les instructions qui utilisent ce bit le précisent dans leur nom. Par exemple ADDX qui se lit add with extend est une opération d'addition prenant en compte ce bit X. Ce bit X est généralement le reflet du bit C, mais, contrairement, à celui-ci, certaines instructions ne le modifient pas.

Lorsque nous étudierons de plus prés les instructions du 68000, le fait que l'instruction affecte ou non tel ou tel bit sera parfois très important.

Le bit T (T signifie Trace donc suivre en Français).
Lorsque ce bit est à 1, le 68000 se trouve en mode Trace.

Alors là, soyez bien attentif, ce qui va suivre est primordial pour la suite des cours!!!

Le mode Trace est un mode de mise au point pour les programmes. Et oui, c'est carrément DANS le microprocesseur qu'une telle commande est insérée. A chaque fois que le 68000 exécute une instruction, il va voir dans quel état se trouve le bit T. S'il trouve ce bit à 0, il passe à la prochaine instruction. Par contre, si ce bit est à 1, le 68000 laisse de côté (temporairement) le programme principal pour se détourner vers une routine (un 'bout' de programme) qui affichera par exemple la valeur de tous les registres (D0 à D7 et A0 à A7). Imaginons qu'il faille appuyer sur une

touche pour sortir de cette routine: Nous avons donc tout le temps de consulter ces valeurs. Nous appuyons sur une touche: fin de notre routine, le 68000 retourne donc au programme principal, exécute l'instruction suivante, teste le bit T, le trouve à nouveau à 1, se branche donc sur notre routine, etc... Nous avons donc un mode pas-à-pas. Or, vous avez déjà utilisé cette particularité en visualisant le déroulement des instructions avec MONST!

Tapez le programme suivant:

```
          MOVE.W     #$23,D0
          MOVE.W     #$15,D1
```

Assemblez et faites Alternate+D pour passer sous MONST. Appuyez une fois sur Control+Z et observez le Status Register. MONST a affiché T, indiquant ainsi que ce bit est à 1. Nous sommes donc bien en mode Trace. Quittez le programme par Control+C.

Nous arrivons maintenant à nous poser une question: Le 68000 a trouvé le bit T à 1. D'accord, il sait où est son Status register et il sait que le bit T c'est le 15ème. Mais après ? Le 68000 s'est détourné vers une routine qui dans le cas présent se trouve être une partie de MONST.

Mais comment a-t-il trouvé cette routine ? MONST est en effet un programme tout à fait ordinaire, qui a été chargé en mémoire à partir de la disquette, et qui peut être placé n'importe où dans cette mémoire.

Une première solution consisterait à toujours placer ce programme au même endroit. MOTOROLA aurait ainsi pu concevoir le 68000 en précisant: Les programmes de mise au point qui seront appelés grâce à la mise à 1 du bit T, devront commencer à l'adresse $5000. Simple, mais très gênant car il devient pratiquement impossible de faire résider plusieurs programmes en mémoire simultanément, sans courir le risque qu'ils se marchent sur les pieds!!!

Il y a pourtant une autre solution, un peu plus tordue mais en revanche beaucoup plus souple, qui consiste à charger le programme de mise au point n'importe où en mémoire, de noter l'adresse à laquelle il se trouve, et de noter cette adresse à un endroit précis. Lorsque le 68000 trouvera le bit T à 1, il foncera à cet endroit prévu à l'avance par MOTOROLA, il y trouvera non pas la routine mais un long mot, adresse de cette routine, à laquelle il n'aura plus qu'à se rendre.

Cet endroit précis, où sera stocké l'adresse de la routine à exécuter lorsque le bit T sera trouvé à 1, c'est un endroit qui se situe dans le premier kilo de mémoire (donc dans les 1024 premiers bytes). En l'occurrence pour le mode trace il s'agit de l'adresse $24.

Résumons: MONST se charge en mémoire. C'est un programme complet dont certaines routines permettent l'affichage des registres. MONST regarde l'adresse à laquelle commencent ces routines, note cette adresse puis va la mettre à l'adresse $24. Ce long mot est donc placé à l'adresse $24, $25, $26 et $27 puisque nous savons que le 'diamètre' du 'tube' mémoire n'est que d'un octet (byte). Lorsque le microprocesseur trouve le bit T à 1, il va à l'adresse $24, il y prélève un long mot qui se trouve être l'adresse des routines de MONST, et il fonce à cette adresse. ok?

Nous allons maintenant réaliser un petit programme et nous allons

'planter' votre ATARI!
Tapez ce qui suit:

```
          MOVE.W     #$1234,D1
          MOVE.W     #$6789,D2
          MOVE.W     #$1122,D3
```

Assemblez puis taper Alternate+D pour passer sous MONST. Faites une fois Control+Z. Le bit T du Status register est mis à 1, indiquant que nous sommes en mode Trace. Comme nous avons exécuté une instruction, D1 se trouve rempli avec $1234. Appuyons maintenant sur Alternate + 3.

Nous venons d'activer la fenêtre de droite (la numéro 3).
Appuyons sur Alternate+A. Une demande s'affiche: nous devons indiquer quelle adresse sera la première visible dans la fenêtre. Il faut taper cette adresse en hexadécimal. Nous tapons donc...24. (pas de $ avant, MONST sait de lui-même que nous parlons en hexa) Nous voyons s'afficher l'adresse 24 en haut de la fenêtre et en face un chiffre qui est l'adresse de notre routine de MONST!

Pour moi c'est 00027086 mais comme je l'ai dit précédemment cela dépend des machines. Dans mon cas lorsque le 68000 trouve le bit T à 1, il fonce donc exécuter la routine qui se trouve en $00027086. Je vais donc modifier cette adresse! Appuyons sur Alternate+E pour passer en mode édition. Le curseur est placé sur le premier nibble de l'adresse. Tapez par exemple 11112222 ou n'importe quel autre chiffre.
Repassez maintenant dans la fenêtre 1 en tapant Alternate+1.

Maintenant réfléchissons: Nous allons refaire Control+Z. Le 68000 va foncer en $24, va maintenant y trouver $11112222, et va foncer à cette adresse pour y exécuter ce qu'il va y trouver c'est-à-dire n'importe quoi! Il y a très peu de chance pour qu'il réussisse à y lire des choses cohérentes et vous indiquera une erreur.

Allez y, n'ayez pas peur, vous ne risquez pas de casser votre machine!

Hop Control+Z et, suivant les cas, vous obtenez divers messages (Illegal exception, Bus Error etc...).

Quittez en faisant Control+C ou bien en dernier ressort faites un RESET.

J'espère que ce principe est TRES TRES BIEN COMPRIS. Si cela vous semble à peu près clair, relisez tout car la suite va très souvent faire référence à ce principe d'adresse dans le premier kilo, contenant l'adresse d'une routine.

La prochaine fois, nous finirons d'étudier le Status Register, en attendant je vais me prendre une petite vodka bien fraîche. A la vôtre!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*            COURS D'ASSEMBLEUR 68000 SUR ATARI ST               *
*                                                                *
*               par Le Féroce Lapin (from 44E)                   *
*                                                                *
*                        Cours numéro 5                          *
*                                                                *
******************************************************************
```

Suite de l'étude du Status register, les interruptions.

Etant donné que nous avons parfaitement compris ce qui se passait dans le cas où le 68000 trouvait le bit T du Status Register à 1, c'est-à-dire tout le système d'adresse fixe à laquelle on trouve l'adresse de la routine, nous allons pouvoir continuer et en fait finir la description des autres bits de ce Status Register.

Le bit S / Superviseur

Le 68000 peut évoluer dans 2 modes: le mode Superviseur et le mode Utilisateur. Dans le mode superviseur, nous avons accès à TOUTES les instructions du 68000 et à TOUTE la mémoire, alors qu'en mode utilisateur certaines instructions ne peuvent être employées, et l'accès à certaines parties de la mémoire est interdit.

Effectivement cela peut sembler au premier abord surprenant: Vous avez acheté une machine, c'est quand même pour pouvoir l'utiliser dans sa totalité! Là encore, nous tombons dans le piège qui consiste à mélanger ATARI ST et 68000 MOTOROLA. Grâce à l'énorme puissance de ce micro-processeur, il est tout à fait possible d'envisager un travail multi-utilisateur.

Gonflons notre ST à 8 Mega octets, équipons le d'un énorme disque dur, et connectons le à plusieurs terminaux. Nous avons donc plusieurs claviers, plusieurs écrans, mais en revanche un seul micro-processeur, celui de l'unité centrale (dont le nom prend ici toute sa valeur) et une seule mémoire, dans laquelle tout le monde pioche à tours de bras. Là, la différenciation Superviseur/Utilisateur prend son sens. Le Superviseur, c'est le 'propriétaire' de l'unité centrale, les personnes utilisant les terminaux n'étant que des 'utilisateurs'. Le fait de ne leur autoriser qu'une partie des instructions et de la mémoire, a pour but d'éviter les plantages car si dans le cas d'une mono-utilisation, un plantage total de la machine est toujours gênant, dans le cas d'une multi-utilisation, cela relève de la catastrophe, car on ne plante plus le travail d'une seule personnes mais de plusieurs!

Le bit S du Status Register, s'il est à 0, indique que nous sommes en mode Utilisateur. A 1, il indique que nous sommes en Superviseur.

Tout comme MONST indiquait l'état Trace en indiquant T à côté du SR, il indique U ou S suivant le mode dans lequel nous nous trouvons.

Jetons un coup d'oeil en arrière sur le chapitre décrivant le brochage du 68000 (cours supplémentaire A). Nous retrouvons cette distinction au niveau des broches FC0, FC1, FC2.

Avant d'étudier les 3 bits restant du SR (I2, I1, I0), il faut savoir que le Status Register est en fait séparé en 2 octets. L'octet de poids fort (bit 8 à 15) est appelé octet superviseur, alors que l'octet de poids faible est l'octet utilisateur.

En mode utilisateur on ne peut écrire que dans l'octet utilisateur alors qu'en mode superviseur nous avons accès au word dans sa totalité.

L'octet utilisateur contenant les bits de conditions (bits X N Z V C), on l'appelle également registre des codes conditions (Condition Code Register), ou CCR.

Les bits I2, I1 et I0 (Interrupt Mask)

Ces 3 bits servent à représenter les masques d'interruption. Mais voyons tout d'abord ce qu'est une interruption. Nous avons étudié précédemment le fonctionnement lié au bit T (trace). Lorsque ce bit est positionné, le programme principal est interrompu, au profit d'une routine spéciale. C'est en quelque sorte le principe de l'interruption.

Une routine en interruption, c'est un bout de programme différent du programme principal. A intervalles réguliers ou à cause d'un élément extérieur, le 68000 va interrompre (c'est bien le mot!) le programme principal, pour aller exécuter cette routine. Lorsque celle-ci sera terminée, il y aura retour au programme principal.

L'exemple le plus simple est celui du téléphone: Je travaille à mon bureau (c'est le programme principal) lorsque le téléphone sonne. Je détecte l'interruption, j'arrête mon travail et je décroche (exécution de l'interruption). La conversation terminée, je raccroche et je retourne à mon occupation première.

Maintenant, plus compliqué: Interruption de mon travail principal. Je décroche, mais en cours de conversation, on sonne à la porte. Là intervient le principe de la priorité d'interruption. Si la porte d'entrée à une priorité supérieure à celle du téléphone, j'interrompt la conversation téléphonique pour aller ouvrir: Il y a interruption de l'interruption. Une fois claqué la porte au 124ème marchand de balayettes de la journée je reprends le téléphone, je finis la conversation, je raccroche puis je me remets à ma tâche principale.

Par contre, si l'interruption 'porte d'entrée' a une priorité inférieure à celle du téléphone, j'attendrai d'avoir fini avec celui-ci avant d'aller ouvrir.

Les 3 bits I2, I1 et I0 (Interrupt mask) permettent de définir le niveau mini d'interruption qui sera pris en cours. Comme on ne possède que 3 bits, on ne peut définir que 7 niveau, de 1 à 7 (on ne parle pas ici du niveau 0, car c'est le niveau de travail 'normal' de la machine. Si le niveau est à 0, c'est qu'il n'y a pas d'interruption.). Ainsi, si nous avons 011 pour ces 3 bits, nous obtenons 3 comme niveau mini. Les interruptions de niveaux 1 et 2 ne seront donc pas prises en compte. Puisque le niveau indiqué par les 3 bits sera accepté comme niveau d'interruption, nous en déduisons que si les bits sont à 111, seuls les interruptions de niveau 7 seront prises en compte. Or nous voyons bien également qu'il n'est pas possible de définir un niveau minimum de 8 par exemple, et donc qu'il sera impossible d'empêcher une interruption de niveau 7. Ce niveau est donc dit 'non-masquable'.

Les interruptions de niveau 7 sont donc appelées NMI c'est à dire non-maskable-interrupt.

A noter qu'il n'est pas possible d'opérer une sélection précise et par exemple d'autoriser les interruptions de niveaux 4, 5 et 7 et pas celles de niveau 6. Si les bits sont à 100, les interruptions de niveau 4, 5, 6 et 7 seront autorisées. Vous pouvez jeter à nouveau un coup d'oeil sur le cours annexe A. Vous retrouverez bien sur le 68000 les broches I2, I1 et I0. Une remarque cependant, ces broches sont actives à l'état bas, c'est-à-dire qu'elle indique quelque chose lorsqu'il n'y a pas de courant, à l'inverse des autres broches.

Par contre leur représentation au sein du Status Register se fait dans le bon 'sens'.
Nous sommes maintenant amenés à nous poser une question similaire à celle que nous nous sommes posée lors de l'étude du mode Trace. Le 68000 reçoit une demande d'interruption. Il compare le niveau de celle-ci à la limite fixée par les bits I du Status Register.

Si l'interruption est acceptable, il sauve le Status Register et met en place dans les bits I le niveau de l'interruption qu'il va exécuter afin de ne pas être gêné par une autre demande plus faible. Il stoppe alors l'exécution de son programme principal pour se détourner vers la routine. Une fois celle-ci terminée, il revient au programme principal. C'est bien joli, mais où a-t-il trouvé la routine en interruption ? Et bien simplement en utilisant le même principe que pour le mode Trace. Nous avons vu que lorsque le bit T était en place, le 68000 allait voir à l'adresse $24 et qu'il y trouvait un long mot, ce long mot étant l'adresse de la routine. Pour les interruptions, le principe est le même: si c'est une interruption de niveau 4, c'est à l'adresse $70 que le 68000 trouvera un long mot, ce long mot, comme dans le cas du mode Trace étant l'adresse de la routine à exécuter. Si l'interruption est de niveau 1, c'est le long mot situé à l'adresse $64 etc... Il est bien évident que c'est au programmeur de placer ces long mots à ces adresses: On prépare une routine, on cherche son adresse de départ, puis on note celle ci à l'endroit précis où l'on sait que le 68000 viendra la chercher.

Toutes ces adresses étant situées dans le premier kilo de mémoire de notre machine, étudions de plus près ces 1024 octets. (Vous trouverez un tableau représentant ce kilo en annexe) Pour le moment nous n'allons faire qu'y repérer les quelques éléments que nous avons déjà étudiés. Toutes ces adresses ont des numéros d'ordres, et à cause de leur fonction propre (ne faire que communiquer l'adresse d'une routine), on les appelle 'vecteurs'.

Nous retrouvons bien en $24 le vecteur 9, correspondant au mode Trace, de $64 à $7C les vecteurs correspondants aux interruptions de niveau 1 à 7. Le niveau 0, étant le niveau 'normal' de travail, n'a pas de vecteur.

Nous pouvons déjà expliquer d'autres vecteurs: Ainsi le numéro 5 (adresse $14) c'est le vecteur de division par 0. Le 68000 ne peut pas faire de division par 0. Lorsque le programme essaye, il se produit la même chose que pour le mode Trace: Ayant détecté une division par 0, le 68000 fonce à l'adresse $14, y trouve une adresse de routine et va exécuter celle-ci. Dans la plupart des cas cette routine va afficher quelques bombes à l'écran et tout bloquer. Rien ne vous empêche cependant de préparer votre propre routine et de mettre son adresse en $14. Ainsi dans un programme de math (beurkk!) cette routine peut afficher "division par 0 im-

possible". Si l'utilisateur tente une telle division, inutile de faire des tests pour le prévenir de cette impossibilité, le 68000 s'en chargera tout seul.

Les autres vecteurs

Erreur bus. Nous avons vu précédemment que le 68000 utilise ce que nous appelons un bus pour recevoir ou transmettre des données. Si une erreur survient sur celui ci, il y a saut a l'adresse $8 pour trouver l'adresse de la routine qui sera alors exécutée.

Erreur d'adresse. Le 68000 ne peut accéder qu'à des adresses paires. S'il tente d'accéder à une adresse impaire, il se produit une erreur d'adresse (même principe de traitement que l'erreur bus, ou le mode Trace, vecteur, adresse etc...). Nous verrons plus tard qu'il nous sera possible d'accéder à des adresse impaires, mais avec des précautions.

Instructions illégales. Nous avons vu que le travail de l'assembleur consistait simplement à transformer en chiffres, ligne par ligne, notre programme. Cependant, si nous mettons en mémoire une image, celle-ci sera également placée dans le 'tube mémoire' sous forme de chiffres. La différence c'est que ces chiffres là ne veulent rien dire pour le 68000 en tant qu'instruction. Si nous ordonnons au 68000 d'aller à cette adresse (celle de l'image) il essayera de décrypter ces chiffres comme des instructions, ce qui déclenchera une erreur 'instruction illégale'.

Violation de privilège. Nous avons vu que le 68000 pouvait évoluer en mode utilisateur ou en mode superviseur. On dit que l'état superviseur est l'état privilégié (ou état de plus haut privilège). Tenter d'accéder en mode utilisateur à une zone mémoire réservée au mode superviseur ou bien tenter d'exécuter une instruction privilégiée (donc utilisable uniquement en superviseur) provoquera une erreur 'violation de privilège'.

Connaître ces différents types d'erreurs est très important. En effet la phase de mise au point est généralement longue en assembleur, surtout au début. De très nombreuses erreurs peuvent survenir, dont la cause est parfois juste sous notre nez. Le type même de l'erreur, si celle-ci est bien comprise, peut souvent suffire à orienter les recherches plus précisément et ainsi raccourcir le temps (pénible) de recherche du grain de sable qui bloque tout!

Tous les vecteurs constituant le premier kilo de mémoire ayant pour but de dérouter le programme principal vers une routine exceptionnelle, sont appelés 'vecteurs d'exceptions'.

Les vecteurs restants seront étudiés dans les séries suivantes, au fur et à mesure des besoins. Chaque chose en son temps!

Pour aujourd'hui nous nous arrêterons là. Ce fut court mais le prochain chapitre sera consacré à la pile et sera bien gros!

La pile est un problème aussi simple que les autres, qui demande simplement de l'attention. Après avoir étudié ce qu'est la pile, il ne nous restera plus qu'un demi-cours avant d'aborder nos premiers 'gros' programmes!

Courage! mais surtout prenez votre temps! Relisez les cours précédent même si tout vous paraît compris. Plus nous avancerons plus le nombre de petites choses augmentera et moins il y aura de place

pour la plus petite incompréhension.

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Cours numéro 6                         *
*                                                                *
******************************************************************
```

LA PILE Nous avons déjà utilisé la notion de 'tube' en ce qui concerne la mémoire. Nous pouvons y stocker différentes choses, et si nous nous rappelons l'adresse, nous pouvons revenir plus tard à cet endroit pour y récupérer ce que nous y avions déposé.

Essayez avec ce petit programme:

```
       MOVE.L     #$12345678,D0
       MOVE.L     D0,$493E0
       MOVE.L     #0,D0
       MOVE.L     $493E0,D0
```

Assemblez puis passez sous MONST. Avancez en pas à pas. D0 est d'abord rempli avec $12345678, puis le contenu de D0 est transféré à l'adresse $493E0. Notez bien qu'il n'y a pas de # devant $493E0, afin d'indiquer qu'il s'agit bien d'une adresse. Cette ligne étant exécutée, activez la fenêtre 3 ([Alternate+3]) et placez le début de celle-ci sur l'adresse $493E0 ([Alternate+A] puis tapez 493E0) Vous voyez bien 12345678 à cet endroit dans le 'tube'. Si j'ai choisit cette adresse c'est parce qu'elle se situe à 300 Kilo du début de la mémoire. Elle est donc accessible même sur un 520, et elle est suffisamment éloignée pour ne pas se trouver dans GENST ou MONST. En effet il n'y a qu'un 'tube' mémoire! Nous sommes donc en train d'écrire dans la mémoire alors qu'une partie de celle-ci est occupée par GENST et MONST! Ecrire à l'intérieur des zones occupées par ces programmes est possible, ce qui entraînera très certainement quelques plantages de ceux-ci!

Continuons en pas à pas, nous mettons D0 à 0 puis le contenu de l'adresse $493E0 (sans #) est remis dans D0.

La pile, c'est une partie de ce tube, mais que nous allons gérer d'une manière un peu différente. En effet, au lieu de placer les données dans le tube et de noter leurs adresses, nous allons cette fois-ci les empiler et pour les récupérer, les dépiler. L'avantage c'est le gain de temps (pas la peine de se demander à quelle adresse on a stocké les données) et un gain de place (si c'est pour stocker temporairement des données, inutile de conserver une portion de 'tube' juste pour ça).

Par contre l'inconvénient c'est que la gestion doit être rigoureuse. Imaginons que j'empile un premier chiffre puis 10 autres

par dessus. Ensuite je dépile, mais erreur de ma part, je ne dépile que 9 chiffres! Quand je dépilerai une fois de plus, croyant retrouver le premier chiffre empilé, je récupérerai en fait le premier de la série de 10.

Nous en concluons 2 choses: d'abord que la pile est un moyen simple pour sauvegarder des données, mais ensuite que c'est une source de désagrément potentiel, tel que certains programmeurs hésite à s'en servir. C'est généralement à cause d'un manque de rigueur ce qui, je l'espère ne vous arrivera pas.

Une autre remarque: le dernier élément placé sur la pile sera toujours le premier à en sortir. C'est bien le même principe que celui d'une pile d'assiettes: Regardez chez vous, il y a sûrement une énorme pile d'assiettes, mais par le simple fait que le rangement après le lavage se fait par empilage et que mettre la table se fait par dépilage, vous mangez en fait toujours dans les mêmes assiettes... (d'où l'intérêt de bien faire la vaisselle!)

Cette structure de pile est appelée structure LIFO, c'est-à-dire Last In First Out, en Français: 'dernier entré premier sorti'. Cette structure est différente d'une autre structure fréquemment rencontrée en informatique, celle de la file, appelée aussi structure FIFO (First In First Out), la file étant similaire à une file d'attente devant un guichet: le premier dans la file sera le premier parti.

Mais concrètement, à quoi sert la pile? Nous allons le voir avec un exemple. Tapez le programme suivant:

```
        MOVE.L     #$12345678,D0
        MOVE.L     #$BD88,D1
        MOVE.L     #$BD88,A0
        BSR        AJOUTE
        MOVE.L     #0,D0
        MOVE.L     D2,D0


AJOUTE  MOVE.L     #$11112222,D2
        ADD.L      D1,D2
        RTS
```

Première remarque: ce programme diffère des précédents par le fait que nous utilisons une étiquette, un label qui se nomme AJOUTE.

Ce mot, 'AJOUTE', doit se trouver tout à gauche, contre le bord de la fenêtre de l'éditeur. Ce n'est pas quelque chose à placer DANS le tube mais bien une marque A COTE du tube.

Autre remarque, les listings en assembleur, contrairement aux listings dans les autres langages sont assez libres au niveau présentation. Il est tout a fait possible de passer des lignes, ce qui est fait ici pour séparer les 2 parties. Les sources assembleur sont bien souvent très longs, et même si cela fait perdre quelques lignes, espacer les modules permet de s'y retrouver plus facilement.

Assemblons puis débuggons. Avançons pas à pas avec Control+Z. Les 3 premières lignes nous sont familières mais pas la quatrième. Celle-ci se lit BRANCH SUB ROUTINE AJOUTE, c'est-à-dire branchement à une subroutine nommée AJOUTE. Pour préciser vers quelle su-

broutine on désire se diriger, son étiquette est précisée. Ici en l'occurrence c'est AJOUTE mais le nom importe peu. Il est tout a fait possible de mettre des noms assez longs et je ne peux que vous conseiller d'éviter dans vos listings les noms du genre X Y, Z ou encore AX1 etc... qui sont quand même moins explicites que DEBUT_IMAGE, NEW_PALETTE ou bien END_GAME.

Maintenant soyez très attentifs: à la lecture de cette instruction de nombreuses choses vont se passer. L'ordre demande donc au 68000 de poursuivre la lecture de ses instructions dans un sous programme dont le début se situe dans le tube, en face de l'étiquette AJOUTE. Cependant il s'agit bien ici d'un sous-programme. Ceci suppose qu'une fois terminé, le 68000 remontera pour exécuter la ligne qui suit BSR AJOUTE, en l'occurrence MOVE.L #0,D0. Question: comment le 68000 saura-t-il où remonter? En effet le propre d'une subroutine est de pouvoir être appelée plusieurs fois et de plusieurs endroits différents et de pouvoir à chaque fois revenir à l'endroit même qui l'a appelé.

Eh bien le 68000 va justement utiliser la pile pour noter ce lieu de retour. Cette pile a bien sur une adresse, où se trouve-t-elle notée? En A7. Et oui, ce registre un peu spécial correspond à la pile.

Mais A7' alors? Et bien c'est aussi une pile, mais réservée au mode Superviseur. Ainsi si nous faisions tourner conjointement 2 programmes, l'un en mode utilisateur et l'autre en superviseur, chacun aurait sa pile.

Avant d'exécuter la ligne BSR AJOUTE, observons attentivement les registres d'adresses et les registres de données.

Nous avons vu que les registres, qu'ils soient de données ou d'adresse, peuvent contenir des nombres codés sur 32 bits. Nous avons vu aussi qu'il existait 2 sortes de nombres pour la machine: ceux se trouvant à l'intérieur du 'tube' et ceux ce trouvant à l'extérieur, CONTRE ce tube, et indiquant une sorte de distance par rapport au début de celui-ci.

Ce second type de nombre est appelé adresse. Or il est tout à fait possible de stocker un nombre représentant une adresse dans un registre de données (D0-D7). Imaginons maintenant que nous devions stocker le score d'un joueur dans le jeu que nous programmons. Ce score va par exemple être placé dans la mémoire (dans le 'tube') à l'adresse $80792.

Mais que se passera-t-il si nous transférons cette adresse pour l'utiliser grâce à A1 par exemple? et bien A1 va prendre la valeur $80792. C'est bien joli, mais ce qui nous intéresse, ce n'est pas ça! Ce que nous voulons modifier, vérifier etc.. c'est ce qu'il y a DANS le tube à cette adresse.

Et bien notre débugger anticipe un peu cette demande. En effet, partant du principe que les nombres stockés en D0-D7 ou A0-A6 peuvent représenter des valeurs d'adresses, il indique à côté des registres, ce qu'il y a dans le tube, à l'adresse indiquée dans le registre.

En ce qui concerne les registres de données, MONST affiche à leur droite la valeur de 8 octets se trouvant dans le tube à l'adresse indiquée dans le registre. Pour les registres d'adresse, ce sont 10 octets qui sont indiqués. Vous remarquez certainement qu'en face du registre D0 (qui doit contenir $12345678 si vous avez fait

correctement avancer le programme), MONST n'a affiché que des étoiles. C'est normal car le nombre $12345678 correspond à un emplacement mémoire qui se serait accessible qu'avec 305 méga de mémoire!!! MONST indique donc qu'il ne peut pas atteindre cette zone mémoire en affichant des étoiles.

Regardons maintenant D1 et A0. Les nombres situés à leur droite montrent la même chose, ce qui est normal puisque les 2 registres D1 et A0 sont remplis avec le même nombre. On dit qu'ils pointent sur l'adresse $BD88. Allons voir en mémoire histoire de vérifier l'affichage. Activez la fenêtre 3 avec Alternate+3. Celle-ci nous affiche le contenu de la mémoire, mais nous sommes loin de $BD88!

Demandons donc que cette adresse soit celle du haut de la fenêtre 3, avec Alternate+A. Tapons cette adresse (BD88). La fenêtre 3 se ré affiche avec en haut l'adresse $BD88. Dans la colonne de droite nous voyons le contenu de la mémoire, dont nous avions déjà un aperçu avec l'affichage à droite de D1 et de A0. C'est clair?

Réactivons la fenêtre 1 (alternate+1). Normalement la petite flèche doit toujours se trouvez en face du BSR AJOUTE. Noter le chiffre se trouvant dans le registre A7 (donc l'adresse de la pile) et observer bien les chiffres à droite de ce registre, tout en faisant Control+Z.

Les chiffres ont changé! D'abord le registre A7 ne contient plus le même nombre. Celui qui s'y trouve actuellement est en effet plus petit que le précédent. Notons que cette différence est de 4. L'adresse de la pile a donc été décrémentée de 4. De plus des chiffres ont été placés dans la pile (on les voit à droite du registre A7). Or, regardez bien le nombre qui est à gauche de l'instruction MOVE.L #0,D0 de notre programme, c'est-à-dire l'adresse à laquelle devra revenir le 68000 une fois la subroutine terminée: c'est bien ce nombre qui a été placé dans la pile. Il y a donc empilage de l'adresse de retour, ce qui explique également le changement d'adresse de la pile de 4. En effet une adresse est codée sur 4 octets !

Note: étant donné que nous parlons de pile, on dit plus souvent que les données sont mises sur la pile et moins souvent dans la pile.

Continuons notre programme avec Control+Z. Nous sommes maintenant dans la sous-routine. Arrêtez juste avant RTS. C'est cette instruction qui va nous faire "remonter". Elle se lit RETURN FROM SUB ROUTINE.

Observons A7 (sa valeur mais aussi le contenu du 'tube' à cette adresse) et faisons un pas (Control+Z). L'adresse de retour a été dépilée, A7 a repris son ancienne adresse et nous pointons maintenant sur MOVE.L #0,D0.

Quittez ce programme avec Control+C, effacez le et tapez celui-ci.

```
        MOVE.L     #$12345678,D0
        MOVE.L     #$AAAAAAAA,D1
        BSR        AJOUTE
        MOVE.W     D2,D3

AJOUTE  MOVE.W     #$EEEE,D1
        MOVE.W     #$1111,D2
        ADD.W      D1,D2
        RTS
```

Assemblez puis débuggez. Avancez pas à pas: D0 prend la valeur $12345678 D1 la valeur AAAAAAAA, puis nous partons vers la subroutine AJOUTE.

Malheureusement celle-ci utilise D1 et au retour nous constatons que celui-ci ne contient plus AAAAAAAA. En effet le branchement à une subroutine ne sauve rien d'autre que l'adresse de retour, et en assembleur les variables locales et autres bidouilles de langages évolués n'existent pas! C'est donc à nous de sauver les registres, et c'est ce que nous allons faire maintenant.

Note: le registre A7 contenant l'adresse du sommet de la pile (cette adresse variant bien sûr avec l'empilage et le dépilage), on peut considérer cette adresse comme un doigt indiquant perpétuellement le sommet de la pile. Pour cette raison le registre A7 est aussi appelé pointeur de pile. Comme toujours nous utiliserons le vocabulaire anglo-saxon, et nous dirons Stack Pointer, en abrégé SP. Pour cette raison et parce que l'usage en est ainsi, nous remplacerons désormais A7 par SP (qui ne se lit pas "èss-pé" mais bien STACK POINTER!!!).

Imaginons que nous voulions sauvegarder D0 à l'entrée de la subroutine:

Il ne faudra pas oublier de le récupérer à la sortie! Déplaçons donc le contenu de D0 vers la pile. Essayons MOVE.L D0,SP et réfléchissons: Ceci va mettre le contenu de D0 dans A7, malheureusement ce n'est pas ce que nous voulons faire. En effet nous désirons mettre le contenu de D0 DANS le tube, à l'endroit indiqué par A7 (donc SP).

Ceci va se faire avec MOVE.L D0,(SP) ,les parenthèses indiquant que la source de l'opération c'est l'intérieur du tube.

Effacez le programme actuel et tapez le suivant.

```
       MOVE.L     #$12345678,D0
       MOVE.L     D0,(A0)
       MOVE.W     D0,(A1)
```

Assemblez puis comme d'habitude débuggez. D0 prend la valeur $12345678, puis D0 est transféré dans sa totalité (à cause du .L qui indique que l'opération se passe sur un mot long) à l'adresse qui est notée dans A0, ensuite le poids faible de D0 est transféré dans le tube à l'adresse notée en A1. Pour le vérifier, vous pouvez activer la fenêtre 3 et demander à placer l'adresse notée dans A0 en haut de cette fenêtre, et vous constaterez qu'effectivement la valeur de D0 se trouve bien dans le 'tube'.

Nous allons donc utiliser ce type de transfert pour sauvegarder D0

Mais réfléchissons encore un peu. MOVE.L D0,(SP) va bien placer le contenu du long mot D0 dans le tube, mais si nous voulons placer une autre valeur sur la pile, celle-ci va écraser notre première valeur car avec MOVE.L D0,(SP) l'adresse indiqué par SP (donc A7) ne va pas être modifiée, ce qui devrait être le cas.

Nous allons donc réaliser le transfert différemment (en fait nous allons encore améliorer notre vocabulaire, puisque nous allons parler maintenant de type ou de mode d'adressage).

Nous allons faire

```
        MOVE.L      D0,-(SP)
```

C'est le mode d'adressage avec pré-décrémentation. Derrière ce vocabulaire pompeux se cache toute une suite d'événements. En une seul instruction, nous diminuons l'adresse du pointeur de pile de 4 (puisque dans notre exemple nous voulions transférer un long mot donc 4 octets), et nous plaçons en mémoire à cette adresse le long mot D0.

Pour récupérer D0, c'est-à-dire dépiler, il faudra faire:

```
        MOVE.L      D0,(SP)+
```

Comme nous avions décrémenté le pointeur de pile pour ensuite déposer D0 à cette adresse, nous récupérons donc D0 sans oublier ensuite de modifier le pointeur de pile dans l'autre sens, pour qu'il retrouve son ancienne position. Notons que dans le cas présent, et si nous nous contentons de réfléchir très sommairement, il aurait été possible de sauver D0 par MOVE.L D0,(SP) et de le récupérer par MOVE.L (SP),D0. C'est compter sans le fait que la pile est un réservoir commun à beaucoup de choses. Il faut donc de préférence jouer à chaque fois le jeu d'un empilage correct et réfléchi mais aussi d'un dépilage 'collant' parfaitement avec l'empilage précédent.

Vérifions tout cela avec l'exemple suivant:

```
        MOVE.L      #$12345678,D0 valeur dans D0
        MOVE.W      #$AAAA,D1      valeur dans D1
        MOVE.L      D0,-(SP)       sauve D0.L sur la pile
        MOVE.W      D1,-(SP)       idem D1 mais en word
        MOVE.L      #0,D0          remet D0 à 0
        MOVE.W      #0,D1          et D1 aussi
        MOVE.W      D1,(SP)+       récupère D1 (word)
        MOVE.L      D0,-(SP)       puis D0
```

Assemblez puis faites défiler ce programme pas à pas sous MONST. Notez plusieurs choses: tout d'abord des commentaires ont été ajoutés au source. Il suffit que ceux-ci soient séparés des opérandes pour que l'assembleur sache qu'il s'agit de commentaires. Si vous désirez taper une ligne de commentaires (c'est-à-dire que sur celle-ci il n'y aura rien d'autre que ce commentaire), vous devez le faire précéder du caractère étoile ou d'un point virgule.

Seconde chose, nous avions empilé D0 puis D1, ensuite nous avons dépilé D1 puis D0. Il faut en effet bien faire attention à l'ordre et aux tailles de ce que nous empilons, afin de pouvoir dépiler les mêmes tailles, dans l'ordre inverse de l'empilage.

Voici un dernier exemple.

```
        MOVE.L      #$12345678,D0
        BSR         AJOUTE      saut vers subroutine
        MOVE.L      D0,D1       transfert

AJOUTE  MOVE.L      D0,-(SP)    sauve d0.l sur la pile
        MOVE.W      #8,D0
        MOVE.W      #4,D1
        ADD.W       D0,D1
        MOVE.L      (SP)+,D0
        RTS
```

Assemblez puis suivez le déroulement sous MONST en étudiant bien le déroulement. Vous voyez bien que le BSR sauve l'adresse de retour sur la pile, puis que D0 et mis par dessus pour être ensuite

récupéré. Ensuite c'est l'adresse de retour qui est reprise et le programme remonte.

Maintenant, provoquons une erreur, une toute petite erreur mais qui sera fatale à notre programme. Au lieu de récupérer D0 par un MOVE.L (SP)+,D0, commettons une faute de frappe et tapons à la place MOVE.W (SP)+,D0.

Assemblez et suivez pas à pas. Au moment de la sauvegarde de D0, ce sont bien 4 octets qui vont être placés sur la pile, modifiant celle-ci d'autant. Malheureusement la récupération ne va re-modifier cette pile que de 2 octets. Au moment où l'instruction RTS va essayer de récupérer l'adresse de retour, le pointeur de pile sera faux de 2 octets par rapport à l'endroit où se trouve réellement cette adresse de retour, et celui-ci va se faire à une adresse fausse. En conclusion: prudence et rigueur!!!!!!

Nous venons donc de voir que la pile était utilisée par le 68000 pour certaines instructions, et qu'elle était bien commode comme sauvegarde.

Il est aussi possible de l'utiliser pour transmettre des données, c'est ce que nous allons voir pour conclure ce chapitre.

Problème: Notre programme principal utilise les registres A0 à A6 et D0 à D6. Il va appeler une subroutine destinée à additionner 2 nombres et à retourner le résultat dans D7. Il faudra donc utiliser 2 registres par exemple D0 et D1 pour travailler dans notre routine, et donc les sauvegarder à l'entrée de celle-ci.

Voici le début du programme.

```
        MOVE.L     #$11111111,D0
        MOVE.L     #$22222222,D1
        MOVE.L     #$33333333,D2
        MOVE.L     #$44444444,D3
        MOVE.L     #$55555555,D4
        MOVE.L     #$66666666,D5
        MOVE.L     #$77777777,D6
```

Les 7 premiers registres sont remplis avec des valeurs bidons, juste pour nous permettre de vérifier leurs éventuelles modifications.

Maintenant il faut placer les 2 nombres que nous désirions additionner, dans un endroit tel qu'ils pourront être récupérés par la routine d'addition. Plaçons donc ces 2 nombres sur la pile.

```
        MOVE.L     #$12345678,-(SP)
        MOVE.L     #$00023456,-(SP)
        BSR        AJOUTE              et en route !
```

Rédigeons maintenant notre subroutine, afin de suivre l'ordre de travail du 68000.

De quoi aurons nous besoin dans cette routine ?

De D0 et de D1 qui vont recevoir les nombres empilés et qui vont nous servir au calcul. Il va nous falloir également un registre d'adresse. En effet, lorsque nous allons dépiler nous allons modifier le pointeur de pile, or nous venons d'effectuer un BSR le 68000 a donc empilé l'adresse de retour sur la pile, et modifier celle-ci va compromettre le retour! Nous allons donc copier

l'adresse de la pile dans A0, et utiliser cette copie.

Note: j'ai décidé d'utiliser D0, D1 et A0 mais n'importe quel autre registre aurait tout aussi bien convenu.

Commençons donc par sauver nos 3 registres.

Cela pourrait se faire par:

```
       MOVE.L     D0,-(SP)
       MOVE.L     D1,-(SP)
       MOVE.L     A0,-(SP)
```

Note: je rappelle que cela se lit move long!

Mais le 68000 possède une instruction très utile dans un pareil cas, qui permet de transférer plusieurs registres d'un coup.

Nous allons donc faire:

```
       MOVEM.L    D0-D1/A0,-(SP)
```

Ce qui se lit: move multiple registers.

Si nous devions transférer de D0 à D5 nous aurions fait :

```
       MOVEM.L  D0-D5,-(SP)
```

et, pour transférer tous les registres d'un seul coup:

```
       MOVEM.L D0-D7/a0-A6,-(SP)        Compris?
```

Sauvons maintenant l'adresse de la pile dans A0. Comme c'est l'adresse qu'il faut sauver et non pas le contenu, cela se fait par:

```
       MOVE.L     A7,A0      transfert du registre A7 vers A0
```

Maintenant nous allons récupérer les 2 nombres que nous avions empilé avant l'instruction BSR.

Imaginons ce qui s'est passé. (A ce propos je vous conseille TRES fortement de vous aider d'un papier et d'un crayon. N'hésitez pas à écrire sur ces cours. Ce sont les vôtres et je ne les réclamerai pas!

Faire un petit dessin ou de placer des pièces sur votre bureau pour vous aider à comprendre est une excellente chose. Bien souvent les manipulations de mémoire ont tendance à devenir abstraites et un petit dessin arrange bien des choses!)

Nous avons décalé de 4 octets le STACK POINTER, puis nous y avons déposé $12345678. Mais dans quel sens avons nous décalé ce SP ?

Vers le début de la mémoire, vers l'adresse 0 de notre tube puisque nous avons fait -(SP). Le pointeur de pile remonte donc le long du tube. Nous avons ensuite recommencé la même chose pour y déposer $23456. Ensuite BSR, donc même chose mais réalisé automatiquement par le 68000 afin d'y déposer l'adresse de retour (4 octets).

Est-ce tout? Non car une fois rendu dans la subroutine nous avons déposé sur la pile les registres D0, D1 et A0. Le transfert ayant été effectué sur le format long mot (MOVEM.L) nous avons transféré

3 fois 4 octets soit 12 octets (bytes).

Notre copie de A7 qui est en A0 ne pointe donc pas sur nos 2 nombres mais beaucoup plus loin. Le nombre que nous avons placé en second sur la pile est donc à 16 vers le début du tube (faites le calcul: 1BSR, + 12 bytes de sauvegarde cela fait bien 16 bytes) et le nombre placé en premier sur la pile suit son copain et se trouve donc à 20 bytes d'ici, en vertu toujours du principe de la pile: le dernier entré, c'est le premier sorti.

Nous pouvons donc dire que $23456 est à A0 décalé de 16 et que $12345678 est à A0 décalé de 20.

Pour récupérer ces 2 nombres plusieurs actions sont possibles:

1) ajouter 16 à l'adresse de A0 puis récupérer.

Une addition d'adresse se fait par ADDA (add adress).

Nous faisons donc

```
       ADDA.L     #16,A0
```

A0 pointe donc maintenant sur $23456, récupérons donc ce nombre et profitons du mode d'adressage pour avancer l'adresse indiquée dans A0 et ainsi tout de suite être prêt pour récupérer l'autre nombre.

```
       MOVE.L     A0)+,D0
```

L'adresse ayant été augmentée nous pouvons donc récupérer la suite:

```
       MOVE.L     (A0)+,D1
```

2) Autre méthode, utilisant un autre mode d'adressage:

La méthode précédente présente un inconvénient: après le ADDA, A0 est modifié et si nous voulions garder cette adresse, il aurait fallu le sauvegarder.

Ou bien nous aurions pu ajouter le décalage à A0, récupérer les données et ensuite retirer le décalage à A0 pour qu'il retrouve sont état de départ.

Autre méthode donc, indiquer dans l'adressage le décalage à appliquer. Cela se fait par:

```
       MOVE.L     16(A0),D0
       MOVE.L     20(A0),D1
```

Cela permet de pointer sur le 16ème octet à partir de l'adresse donnée par A0 et ensuite de pointer sur le 20ème par rapport à A0. Dans les 2 cas, A0 n'est pas modifié.

Voilà le listing complet de cet exemple.

```
       MOVE.L     #$11111111,D0         initialisation de D0
       MOVE.L     #$22222222,D1          idem
       MOVE.L     #$33333333,D2          idem
       MOVE.L     #$44444444,D3         idem
       MOVE.L     #$55555555,D4          idem
       MOVE.L     #$66666666,D5          idem
       MOVE.L     #$77777777,D6         idem
       MOVE.L     #$12345678,-(SP)       passage nombre 1 dans la pile
       MOVE.L     #$00023456,-(SP)       passage nombre 2 dans la pile
       BSR        AJOUTE                 et en route !
       MOVE.L     D7,D0                  transfert du résultat pour
```

voir..

```
* notre subroutine
UTE     MOVEM.L    D0-D1/A0,-(SP)        sauvegarde
        MOVE.L     A7,A0                copie de SP en A0
        MOVE.L     16(A0),D0            récupère 23456 et le met en D0
        MOVE.L     20(A0),D1            récupère 12345678 en D1
        ADD.L      D0,D1                addition
        MOVE.L     D1,D7                transfert du résultat
        MOVEM.L    (SP)+,D0-D1/A0       récupération
        RTS                             et retour
```

* Note: ce programme n'ayant pas de fin 'normale', lorsque vous* serez rendu au retour de la subroutine c'est-à-dire après la ligne" MOVE.L D7,D0 ", quittez le avec Control+C, Assemblez et suivez bien TOUT le déroulement.

Bien sûr, il aurait été possible de faire cela tout différemment. Par exemple nous aurions pu éviter de travailler avec A0. En effet 16(A0) et 20(A0) ne modifiant pas A0, il aurait été plus simple de faire 16(A7) et 20(A7) au lieu de recopier d'abord A7 en A0. De même il aurait été possible de transférer $23456 directement en D7 et $12345678 en D1 puis de faire ADD.L D1,D7 afin d'éviter la sauvegarde de D0 (qui aurait été inutilisée), et le transfert D1 vers D7 qui n'aurait alors pas eu lieu d'être. De même nous aurions pu retourner le résultat par la pile au lieu de le faire par D7.

Beaucoup de variantes possibles n'est ce pas ?

Pour terminer, un petit exercice. Relancer ce petit programme et analysez PARFAITEMENT TOUT ce qui s'y passe. Quelque chose ne va pas ! Je vous aide en disant qu'il s'agit bien sûr de la pile. Cherchez et essayez de trouver comment faire pour arranger ça.

La réponse sera au début du prochain cours mais essayez d'imaginer que c'est votre programme et qu'il ne marche pas et cherchez!!!

Bon, le cours sur la pile se termine ici. Ce fut un peu long mais je pense, pas trop compliqué. Relisez le, car la pile est un truc délicat dont nous allons nous servir TRES abondamment dans le prochain cours. Si vous avez à peu près tout compris jusqu'ici il est encore temps de rattraper le temps perdu et de tout reprendre au début, car il faut avoir PARFAITEMENT tout compris et pas à peu prés!

Afin de vous remonter le moral, je vous signale que vous êtes presque à la moitié du cours...

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
```

```
*                                                                  *
*                    par Le Féroce Lapin (from 44E)                *
*                                                                  *
*                          Cours numéro 7                          *
*                                                                  *
********************************************************************
```

Nous abordons maintenant le septième cours de la série. La totalité du cours étant en 2 séries (enfin à l'heure où je tape ces lignes c'est ce qui est prévu!), celui-ci est le dernier de la première!

A la fin de celui-ci et si vous avez très attentivement et très scrupuleusement suivi les 6 cours précédents, vous devriez être capable d'afficher des images, sauver des fichiers etc...

Mais tout d'abord revenons à notre pile et à la question du cours précédent. Avez vous trouvé l'erreur ?

Eh bien regardez la valeur de A7 avant d'y empiler $12345678 et $23456, et comparez à la valeur à la sortie du programme. Malheur! ce n'est pas la même! Normal, si nous comptons les empilages et les dépilages, nous nous rendons compte que nous avons empilé 8 octets de plus que nous n'avons dépilé. En effet, comme nous avons récupéré nos 2 nombres en sauvegardant au préalable A7 dans A0, nous n'avons pas touché A7 au moment de la récupération. Heureusement d'ailleurs car le retour de la routine aurait été modifié!

Partant du principe de dépilage dans l'ordre inverse, il nous faut donc corriger la pile une fois revenu de la subroutine. Comme nous avons empilé en faisant -(SP) il faut ajouter pour que la pile redevienne comme avant. Ayant empilé 2 nombres de 4 octets chacuns, nous devons ajouter 8 octets à l'adresse de la pile pour la corriger comme il faut. Nous avons déjà vu comment augmenter une adresse, avec ADDA.

Il convient donc de rajouter juste après la ligne BSR AJOUTE une addition sur SP, en faisant ADDA.L #8,SP (qui se lit ADD ADRESS LONG 8 STACK POINTER)

Un appel à une subroutine en lui passant des paramètres sur la pile sera donc typiquement du genre:

```
       MOVE.W     #$1452,-(SP)
       MOVE.L     #$54854,-(SP)
       MOVE.L     #TRUC,-(SP)
       BSR        BIDOUILLE
       ADDA.L     #10,SP
```

Nous passons le word de valeur $1452 dans la pile (modifiée donc de 2 octets), le long mot de valeur $54854 dans la pile (modifiée de 4 octets), l'adresse repérée par le label TRUC dans la pile (modifiée de 4 octets) puis nous partons vers notre subroutine. Au retour correction de 2+4+4=10 octets du stack pointer pour revenir à l'état d'origine.

La pile possède une petite particularité. Nous avons vu dans les cours précédents que le 68000 était un micro-processeur 16/32 bits. Il lui est très difficile d'accéder à des adresses impaires. Or si nous commençons à empiler des octets et non plus uniquement des words ou des long words, le Stack Pointer peut très facilement

pointer sur une adresse impaire, ce qui risque de planter notre machine.

Taper le programme suivante:

```
        MOVE.L     #$12345678,D0
        MOVE.L     D0,-(SP)
        MOVE.B     D0,-(SP)
        MOVE.L     #$AAAAAAAA,D1
```

Assemblez puis passez sous MOnst et avancez pas à pas en observant bien l'adresse du SP (donc celle visible en A7).

Nous constatons que le pointeur de pile se modifie bien de 4 lorsque nous faisons MOVE.L   D0,-(SP) mais qu'il se modifie de 2 lorsque nous faisons MOVE.B   D0,-(SP) alors que nous pouvions nous attendre à une modification de 1 ! Les erreurs provoqués par des adresses impaires sont donc écartées avec la pile . Merci Monsieur MOTOROLA!

(Note: ceci est une particularité des registres A7 et A7'. Si nous avions travaillé avec A3 par exemple au lieu de SP, celui-ci aurait eu une adresse impaire. C'est le type d'usage qui est fait de la pile qui a conduit les gens de MOTOROLA à créer cette différence.)

Abordons maintenant l'ultime chapitre de cette première série:

LES 'TRAP'

Une instruction TRAP est comparable à une instruction BSR. Elle agit comme un branchement vers une routine. Cependant, contrairement à l'instruction BSR qui demande à être complétée par l'adresse, c'est-à-dire le label permettant de trouver la routine, l'instruction TRAP se contente d'un numéro. Ce numéro peut varier de 0 à 15. Lorsque le 68000 rencontre une instruction TRAP il regarde son numéro et agit en conséquence. Vous vous rappeler des tout premiers cours, dans lesquels nous avions parlé du principe utilisé par le 68000 lorsqu'il trouvait la bit T (mode trace) du SR (status register) à 1 ? Saut dans le premier kilo de mémoire (table des vecteurs d'exceptions), recherche de l'adresse $24, on regarde dans le tube à cette adresse, on y trouve un long mot, ce long mot c'est l'adresse de la routine et on fonce à cette adresse exécuter cette routine.

Et bien regardez la feuille qui donne la liste des vecteurs d'exceptions, et jetez un coup d'oeil aux vecteurs 32 à 47. Les voilà nos vecteurs TRAP !!! Lorsque le 68000 rencontre par exemple l'instruction TRAP #8, il fonce à l'adresse $0A0 pour y trouver l'adresse de la routine qu'il doit exécuter.

A priori cela semble bien compliqué pour pas grand chose! En effet il faut prévoir sa routine, la mettre en mémoire, puis placer son adresse dans le vecteur. Plus compliqué qu'un BSR, surtout que BSR REGLAGE_CLAVIER   et plus parlant qu'un TRAP #5 ou un TRAP #12 !!!

Là, nous retournons encore en arrière (je vous avais bien dit que TOUT était important dans ces cours!!!!!) pour nous souvenir de la notion de mode Utilisateur et de mode Superviseur. Le Superviseur accède à toute la mémoire et à toutes les instructions, pas l'Utilisateur.

S'il s'agit d'interdire à l'Utilisateur des instructions assem-

bleur telles que RESET, notre Utilisateur ne sera pas trop gêné par contre c'est en ce qui concerne la mémoire que tout va très sérieusement se compliquer. Voulez vous connaître la résolution dans laquelle se trouve votre machine ? C'est facile, c'est noté à l'adresse $FF8260.

Vous voulez changer la palette de couleur ? Rien de plus simple, elle est notée en $FF8240. Imprimer un petit texte ? A l'aise, il suffit d'employer les registres de communications vers l'extérieur du chip son (étonnant n'est ce pas!). C'est situé en $FF8800 et $FF8802.

Pardon ??? Quoi ??? Vous êtes Utilisateur ??? Ah bon.... Parce que c'est gênant... Toutes ces adresses sont situées dans la zone mémoire uniquement accessible au Superviseur.....

L'Utilisateur se trouve bien coincé et les possibilités s'en trouvent drôlement réduites. Heureusement, les TRAP sont là !!! Grâce à ce système l'utilisateur va avoir accès à des zones qui lui sont normalement interdites. Pas directement, bien sûr, mais grâce au superviseur. Le superviseur a, en effet, fabriqué des routines qu'il a placé en mémoire et dont les adresses sont dans les vecteurs TRAP. Ces routines sont exécutés en mode superviseur et tapent à tour de bras dans les zones mémoires protégées. Lorsque l'Utilisateur veut les utiliser il les appelle par les TRAP. La protection est donc bien assurée car l'Utilisateur ne fait que déclencher une routine dont généralement il ne connaît que les paramètres à lui passer et le type de message qu'il aura en réponse. C'est de cette manière que nous pouvons accéder au système d'exploitation de notre Atari !!!

Petit rappel: qu'est ce qu'un système d'exploitation ?

Le premier qui répond c'est GEM se prend une paire de claques. GEM c'est l'interface utilisateur et pas le système d'exploitation.

Le système d'exploitation (ou Operating System) dans notre cas c'est TOS. La confusion entre interface Utilisateur et système d'exploitation vient du fait que certains systèmes d'exploitation intègrent également un interface utilisateur: c'est par exemple le cas sur PC avec MS DOS.

Le système d'exploitation c'est un ensemble de routine permettant d'exploiter la machine. Ces multiples routines permettent par exemple d'afficher un caractère à l'écran d'ouvrir un fichier, de formater une piste de disquette, d'envoyer un octet sur la prise MIDI etc... En fait tous les 'trucs' de base, mais jamais de choses compliquées. Une routine du système d'exploitation ne permettra pas, par exemple, de lire le contenu d'un fichier se trouvant sur la disquette. En effet ceci demande plusieurs opérations avec à chaque fois des tests:

Ouverture du fichier: existe t-il,
la disquette n'est elle pas abîmée etc...
positionnement du pointeur dans le fichier: le positionnement s'est il bien passé?

Lecture: N'as t-on pas essayé de lire trop d'octets etc, etc....

Il faudra donc bien souvent plusieurs appels à des routines différentes pour réaliser ce que l'on veut.

Il est toujours possible de se passer du système d'exploitation,

spécialement lorsque l'on programme en assembleur. En effet l'ensemble des routines de l'OS (abréviation de Operating System) est destiné à un usage commun, tout comme d'ailleurs les routines de l'interface Utilisateur.

Ceci explique bien souvent la ré-écriture de toutes petites parties du système afin de n'utiliser que le strict nécessaire. La routine de gestion souris du GEM par exemple doit s'occuper de la souris mais aussi du clavier, du MIDI et du joystick. Pour un jeu il peut être intéressant de ré-écrire cette routine afin de gérer uniquement le joystick et donc d'avoir une routine qui 'colle' plus au besoin.

Nous verrons beaucoup plus tard comment regarder dans le système d'exploitation afin de pouvoir par la suite réaliser soi-même ses routines. Avant cela, utilisons simplement ce système!

Nous allons donc l'appeler grâce aux TRAPs.
4 traps sont accessibles 'normalement' dans le ST:

```
TRAP  #1      routines du GEMDOS
TRAP  #2      routines du GEM
TRAP  #13     routines du BIOS
TRAP  #14     routines du BIOS étendu (eXtended Bios donc XBIOS)
```

```
GEMDOS   =Graphic environment manager disk operating system
GEM      =Graphic  environment manager (se découpe par la suite en
AES, VDI  etc.. Un  chapitre de la seconde série y sera consacrée)
BIOS     =Basic Input Output System
XBIOS    =Extended Basic Input Output System
```

Les autres vecteurs TRAP (0, 3 à 12 et 15) sont, bien entendu, actifs mais aucune routine n'y est affectée. Nous pouvons les utiliser pour peu que nous y mettions avant nos routines, ce qui sera l'objet du premier cours de la seconde série.

Nous constatons que le TRAP #1 permet d'appeler le GEMDOS. Or il n'y a pas qu'une routine GEMDOS mais une bonne quantité. De plus ces routines demandent parfois des paramètres. Comment faire pour les transmettre ? Et bien tout simplement par la pile !!!

Taper le programme suivant:

```
       MOVE.W     #65,-(SP)
       MOVE.W     #2,-(SP)
       TRAP       #1
       ADDQ.L     #4,SP

       MOVE.W     #7,-(SP)
       TRAP       #1
       ADDQ.L     #2,SP
       MOVE.W     #0,-(SP)
       TRAP       #1
       ADDQ.L     #2,SP
```

Assemblez ce programme mais ne le débuggez pas, lancez le par Alternate+ X. Vous voyez apparaître un A sur l'écran de votre ST. Appuyer sur une touche et hop vous revenez dans GENST! Analysons ce que nous avons fait car là de très très nombreuses choses se sont passées, et avouons le, nous n'avons rien vu !!!!!

Tout d'abord nous avons appelé la fonction Cconout() du Gemdos. Nous avons appelé le Gemdos avec le TRAP #1, mais cette instruc-

tion nous a envoyé vers un ensemble de routine, toutes appartenant au Gemdos. Pour indiquer à cette routine principale vers quelle subroutine du Gemdos nous désirons aller, nous avons passé le numéro de cette subroutine dans la pile. Partant toujours du principe du dernier entré premier sorti, il est bien évident que ce numéro doit se trouver empilé en dernier afin de pouvoir être dépilé en premier par la routine principale de Gemdos, afin qu'elle puisse s'orienter vers la sous-routine qui nous intéresse. La fonction Cconout ayant le numéro 2, nous avons donc fait MOVE.W #2,-(SP) .(voir plus haut pour se rappeler que 2 peut très bien être codé sur un octet mais, comme nous travaillons vers la pile, il sera pris comme un word de toutes façons).

Maintenant le Gemdos ayant trouvé 2 comme paramètre, s'oriente vers cette routine au nom barbare, qui a pour fonction d'afficher un caractère sur l'écran. Une fois rendu vers cette routine, le Gemdos va chercher à savoir quel caractère afficher. C'est pour cela que nous avons placé le code ASCII de ce caractère sur la pile avec MOVE.W #65,-(SP).

Note: Pour l'assembleur, le code ASCII peut être remplacé par la lettre elle-même. Nous aurions donc pu écrire MOVE.W #"A",-(SP) sans oublier toutefois les guillemets!

De retour du TRAP nous devons corriger la pile, afin d'éviter le problème qui a fait l'objet du début de ce cours. Nous avions empilé un word donc 2 octets et ensuite un autre word soit au total 4 octets. Nous allons donc ajouter 4 au SP. Nous profitons ici d'une opération d'addition plus rapide que ADDA, ADDQ qui se lit add quick. Cette addition est autorisée jusqu'à 8 inclus. Il n'est pas possible par exemple de faire ADDQ.L #12,D1

Ensuite nous recommençons le même genre de chose, avec la fonction 7 du GEMDOS (nommée Crawcin)qui elle n'attend aucun paramètre, c'est pourquoi nous passons juste son numéro sur la pile. Cette fonction attend un appui sur une touche. Ayant passé un paramètre sur un word, nous corrigeons au retour du TRAP la pile de 2.

Le programme se termine avec la fonction 0 du GEMDOS (Ptermo) qui libère la mémoire occupée par notre programme et le termine pour de bon. Cette routine n'attend pas de paramètre, nous ne passons dans la pile que son numéro donc correction de 2. Note: la correction de pile pour la fonction Ptermo n'est là que par souci pédagogique. Cette fonction terminant le programme, notre dernière instruction ADDQ.L #2,SP ne sera jamais atteinte!

Plusieurs choses maintenant. D'abord ne soyez pas étonnés des noms bizarres des fonctions du GEMDOS, du Bios ou du Xbios. Ce sont les véritables noms de ces fonctions. En assembleur nous ne les utiliserons pas directement puisque l'appel se fait pas un numéro, mais en C par exemple c'est ainsi que sont appelées ces fonctions. Dans les cours d'assembleur de ST MAG (dont les vertus pédagogiques sont plus que douteuses), nous pouvons lire que les noms de ces fonctions ont été choisis au hasard et que la fonction Malloc() par exemple aurait pu s'appeler Mstroumph(). C'est ridicule! Chacun des noms est, comme toujours en informatique, l'abréviation d'un expression anglo-saxonne qui indique concrètement le but ou la fonction. Ainsi Malloc signifie Memory Allocation, cette fonction du GEMDOS permet donc de réserver une partie de mémoire!!! Malheureusement de nombreux ouvrages passe sur ce 'détail' et ne fournissent que l'abréviation.

Ceci n'empêche qu'il vous faut impérativement une liste de toutes

les fonctions du GEMDOS, du BIOS et du XBIOS. Ces fonctions sont décrites dans le Livre du Développeur, dans la Bible mais également dans les dernières pages de la doc du GFA 3.

Note: dans la doc du GFA, il manque la fonction GEMDOS 32 qui permet de passer en Superviseur. Ce mode n'étant pour le moment que d'un intérêt limité pour vous, pas de panique, nous décrirons tout cela dans la seconde série.

Continuons pour le moment avec des petits exemples.
Affichons une phrase sur l'écran à la place d'un lettre.
Ceci va se faire avec la programme suivant:

```
          MOVE.L     #MESSAGE,-(SP)        adresse du texte
          MOVE.W     #9,-(SP)   numéro de la fonction
          TRAP       #1         appel gemdos
          ADDQ.L     #6,SP      correction pile

   * attente d'un appui sur une touche

          MOVE.W     #7,-(SP)   numéro de la fonction
          TRAP       #1         appel GEMDOS
          ADDQ.L     #2,SP      correction pile

   * fin du programme

          MOVE.W     #0,-(SP)
          TRAP       #1

          SECTION DATA

MESSAGE   DC.B       "SALUT",0
```

Une nouveauté, le passage d'une adresse. En effet la fonction 9 du gemdos demande comme paramètre l'adresse de la chaîne de caractère à afficher. Nous avons donc donné MESSAGE, qui est le label, l'étiquette servant à repérer l'emplacement dans le tube où se trouve notre phrase, tout comme nous avions mis une étiquette AJOUTE pour repérer notre subroutine, dans le cours précédent.

Ce message est une suite de lettres, toutes codées sur un octets. Pour cette raison nous disons que cette chaîne est une constante constituée d'octet. Nous définissons donc une constante en octets: Define Constant Byte, en abrégé DC.B Attention ceci n'est pas une instruction 68000 ! C'est simplement une notation pour l'assembleur afin de lui dire:

n'essaye pas d'assembler ça comme du code normal, ce n'est qu'une constante. De même nous définissons une zone.

La fonction 9 du GEMDOS demande à ce que la phrase se termine par 0, ce qui explique sa présence à la fin.

Réalisons maintenant un programme suivant le schéma suivant:

affichage d'un texte de présentation en inverse vidéo;

ce texte demande si on veut quitter ou voir un message

si on choisit quitter, bye bye

sinon on affiche 'coucou' et on redemande etc...

Détaillons un peu plus, en traduisant ce programme en pseudo-code. C'est ainsi que l'on nomme la façon de présenter un déroulement d'opération en langage clair mais dont l'organisation se rapproche déjà de la programmation.

```
AFFICHE        "QUITTER (Q) OU VOIR LE MESSAGE (V) ?"
SI REPONSE=Q
     VA A QUITTER
SI REPONSE=V
     AFFICHE "COUCOU"
     RETOURNE A AFFICHE "QUITTER...."
SI REPONSE DIFFERENTE RETOURNE A AFFICHE "QUITTER..."
```

Par commodité, ce listing se trouve sur une feuille séparée (listing numéro 1 / Cours numéro 7).

Tout d'abord affichage de la phrase qui servira de menu, avec la fonction Gemdos 9. Cette phrase se trouve à l'étiquette MENU, allons la voir pour la détailler. Nous remarquons tout d'abord qu'elle commence par 27. Après avoir regardé dans une table de code ASCII, nous notons qu'il s'agit du code ASCII de la touche Escape. Nous cherchons donc d'abord à afficher Escape. Mais, comme vous le savez sûrement, ce caractère n'est pas imprimable!

Impossible de l'afficher à l'écran!

C'est tout à fait normal! en fait il n'est pas question ici d'afficher réellement un caractère, mais plutôt de faire appel à un ensemble de routines, répondant au nom de VT52. Pour appeler ces routines, il faut afficher Escape. Voyant cela le système se dit: "Tiens, on cherche à afficher Escape, c'est donc en fait que l'on cherche à appeler le VT52".

L'émulateur VT52 réagit donc, mais que doit-il faire ? et bien pour le savoir il va regarder la lettre qui suit Escape. En l'occurrence il s'agit ici de E majuscule. Regardez dans les feuilles annexes à cette série de cours, il y en a une consacrée au VT52. Nous voyons que Escape suivi de E efface l'écran, c'est donc ce qui va se passer ici.

Ensuite il était dit dans le 'cahier des charges' de notre programme, que le MENU devait être affiché en inverse vidéo.

Consultons donc la feuille sur le VT52. Nous y trouvons: Escape et 'p' minuscule = passe en écriture inverse vidéo. Juste ce qu'il nous faut! Nous remettons donc 27,"p" dans notre phrase.

Trois remarques:

tout d'abord il faut remettre à chaque fois Escape. Faire 27,"E","p" aurait effacé l'écran puis aurait affiché p.

Seconde remarque, il faut bien faire la différence entre les lettres majuscules et les lettres minuscules. Escape+E efface l'écran mais Escape+e active le curseur!!!

Troisième remarque, on peut représenter dans le listing une lettre par son 'caractère' ou bien par son code ASCII.

Ainsi si on veut afficher Salut, on peut écrire le listing comme ceci:

```
TXT       DC.B       Salut",0
```

ou bien comme cela:

```
TXT    DC.B    83,97,108,117,116,0
```

Il est de même possible de mélanger les données en décimal , en binaire, en hexadécimal et les codes ASCII. Par exemple ceci:

```
TXT       DC.B       65,$42,%1000011,"D",0
```

affichera ABCD si on utilise cette "phrase" avec Gemdos 9.

Ceci vous sera bien utile lorsque vous chercherez à afficher des lettres difficiles à trouver sur le clavier. Pour le 'o' tréma, il est possible de faire:

```
TXT       DC.B       "A bient",147,"t les amis.",0
```

Note: J'espère que depuis le début, il n'y en a pas un seul à avoir lu DC.B "décébé"!!!! Je vous rappelle que cela se lit Define Constant Byte.

Continuons l'exploration de notre programme. Notre phrase efface donc l'écran puis passe en inverse vidéo. Viens ensuite le texte lui-même:

QUITTER (Q) OU VOIR LE MESSAGE (V) ?

Ensuite une nouvelle commande VT52 pour repasser en vidéo normale, puis 2 codes ASCII qui, eux non plus, ne sont pas imprimables. Ce sont les codes de retour chariot. Le curseur va donc se retrouver tout à gauche de l'écran, une ligne plus bas. Enfin le 0 indiquant la fin de la phrase.

Une fois le 'menu' affiché, nous attendons un appui sur une touche avec la fonction Gemdos numéro 7. Cette fonction renvoi dans D0 un résultat. Ce résultat est codé sur un long mot, comme ceci:

Bits 0 à 7 code ASCII de la touche
Bits 8 à 15 mis à zéro
Bits 16 à 23 code clavier
Bits 24 à 31 Indication des touches de commutation du clavier (shifts..)

Dans notre cas nous ne nous intéresserons qu'au code ASCII de la touche enfoncée. Nous allons donc comparer le word de D0 avec chacun des codes ASCII que nous attendons, c'est à dire Q, q, V et v. Cette comparaison va se faire avec une nouvelle instruction: Compare (CMP). Comme nous comparons un word nous notons CMP.W, que nous lisons COMPARE WORD. Nous comparons Q avec D0 (nous aurions pu marquer CMP.W #81,D0 puisque 81 est le code ASCII de Q).

Cette comparaison effectuée, il faut la tester. Nous abordons ici les possibilités de branchement dépendant d'une condition, c'est-à-dire les branchements conditionnels.

Chacune de ces instructions commence par la lettre B, signifiant BRANCH. En clair, ces instructions peuvent être lues comme:

Va à tel endroit si...

Mais si quoi ???

Eh bien plusieurs conditions sont disponibles, que l'on peut regrouper en 3 catégories:

D'abord une catégorie qui réagit à l'état d'un des bits du Status

Register:

```
  BCC  Branch if carry clear (bit de retenue à 0)
  BCS  Branch if carry set   (bit de retenue à 1)
  BNE  Branch if not equal   (bit de zéro à 0)
  BEQ  Branch if equal       (bit de zéro à 1)
  BVC  Branch if overflow clear (bit de dépassement à 0)
  BVS  Branch if overflow set   (bit de dépassement à 1)
  BPL  Branch if plus  (bit négatif à 0)
  BMI  Branch if minus (bit négatif à 1)
```

Une seconde catégorie, réagissant à la comparaison de nombres sans signe.

```
  BHI  Branch if higher  (branche si supérieur à)
  BLS  Branch if lower or same (inférieur ou égal)
(on peut aussi remettre BEQ et BNE dans cette catégorie)
```

UNe troisième catégorie, réagissant à la comparaison de nombres avec signe.

```
  BGT Branch if greater than (si supérieur à)
  BGE Branch if greater or equal (si supérieur ou égal à)
  BLT Branch if lower than  (si plus petit que)
  BLE Branch if lower or equal (si plus petit ou égal)
 (on peut encore remettre BEQ et BNE!!!)
```

Je suis profondément désolé pour les gens de MICRO-APPLICATION (Le Langage Machine sur ST, la Bible, le Livre du GEM etc...) ainsi que pour le journaliste qui écrit les cours d'assembleur dans STMAG, mais les branchements BHS et BLO, malgré le fait qu'ils soient acceptés par de nombreux assembleurs, N'EXISTENT PAS!!!!!

Il est donc impossible de les trouver dans un listing assemblé, l'assembleur les convertissant ou bien les rejetant.

Cet ensemble de branchement conditionnel constitue un ensemble de commande du type Bcc (branch conditionnaly)

Poursuivons notre lente progression dans le listing...
La comparaison est effectuée, testons la:

```
       CMP.W      #"Q",D0    est-ce la lettre 'Q' ?
       BEQ        QUITTER    branch if equal 'quitter'
```

C'est à dire, si c'est égal, sauter à l'étiquette QUITTER.
Si ce n'est pas égal, le programme continue comme si de rien n'était, et tombe sur un nouveau test:

```
       CMP.W      #"q",D0    est-ce q minuscule ?
       BEQ        QUITTER    branch if equal quitter
```

Nous comparons ensuite à 'V' majuscule et en cas d'égalité, nous sautons à AFFICHAGE. Viens ensuite le test avec 'v' minuscule. Là, c'est l'inverse: Si ce n'est pas égal, retour au début puisque toutes les possibilités ont été vues. Par contre, si c'est 'v' qui a été appuyé, le programme continuera sans remonter à DEBUT, et tombera de lui même sur AFFICHAGE.

L'affichage se fait classiquement avec Gemdos 9. Cet affichage terminé, il faut remonter au début. Ici, pas besoin de test car il

faut absolument remonter. Nous utilisons donc un ordre de branchement sans condition (inconditionnel) qui se lit BRANCH ALWAYS (branchement toujours)et qui s'écrit BRA.

En cas de choix 'Q' ou 'q', il y a saut à QUITTER et donc à la fonction Gemdos 0 qui termine le programme.

N'hésitez pas à modifier ce programme, à essayer d'autres tests, à jouer avec le VT52, avant de passer au suivant.

("Quelques heures passent..." In ('Le manoir de Mortevielle') acte 2 scène III)

Prenons maintenant le listing numéro 3. Nous étudierons le numéro 2 en dernier à cause de sa longueur un peu supérieure.

Le but de ce listing est de réaliser un affichage un peu comparable à celui des horaires dans les gares ou les aéroports: chaque lettre n'est pas affichée d'un coup mais 'cherchée' dans l'alphabet.

D'abord effacement de l'écran en affichant Escape et 'E' avec Gemdos 9: rien que du classique pour vous maintenant!

Ensuite cela se complique. Nous plaçons l'adresse de TXT_FINAL dans A6. Regardons ce qu'il y a à cette étiquette 'TXT_FINAL': nous y trouvons la phrase à afficher.

Observons maintenant TRES attentivement ce qui se trouve à l'adresse TXT. Nous y voyons 27,"Y",42 . En regardant notre feuille du VT52 nous voyons que cela correspond à une fonction plaçant le curseur à un endroit précis de l'écran. Nous constatons aussi 2 choses:

1) La commande est incomplète
2) Une phrase affichée par exemple avec gemdos 9, doit se terminer par 0, ce qui ici n'est pas le cas !
   En effet, la phrase est incomplète si on se contente de lire cette ligne. Jetons un coup d'oeil sur la ligne suivante. Nous y trouvons 42, qui est peut être la suite de la commande (nous avons donc escape+Y+42+42), et une ligne encore plus bas nous trouvons deux zéros. Nous pouvons remarquer également que si la phrase commence à l'étiquette TXT, la seconde ligne possède également une étiquette ('COLONE') ainsi que la troisième ligne ('LETTRE').

Imaginons maintenant que nous ayons une lettre à la place du premier zéro en face de l'étiquette LETTRE. Si nous affichons cette phrase nous verrons s'afficher cette lettre sur la 10ème colonne de la 10ème ligne (révisez la commande Escape+Y sur la feuille du VT52).

Imaginons ensuite que nous ajoutions 1 au chiffre se trouvant à l'étiquette COLONNE et que nous recommencions l'affichage. Nous verrions notre lettre toujours 10ème ligne, mais maintenant 11ème colonne!
C'est ce que nous allons faire, en compliquant d'avantage. Plaçons le code ASCII 255 (c'est le code maximale autorisé puisque les codes ASCII sont codés sur un byte) à la place du premier zéro de l'étiquette LETTRE. Nous faisons cela par MOVE.B #255,LETTRE. Ajoutons 1 ensuite au chiffre des colonnes avec ADD.B #1,COLONNE ensuite posons nous la question suivante: la lettre que je vais afficher (actuellement de code ASCII 255), est-ce la même que

celle de la phrase finale ? Pour le savoir il faut prélever cette lettre de cette phrase. Comme nous avons placé l'adresse de cette phrase dans A6, nous prélevons tout en faisant avancer A6 pour pointer sur la seconde lettre. MOVE.B (A6)+,D6

Et si la lettre que nous venons de prélever était le code ASCII 0? Cela voudrais donc dire que nous sommes à la fin de la phrase et donc qu'il faut s'en aller!!! Nous comparons donc D6 qui contient le code ASCII de la lettre, avec 0.

```
        CMP.B       #0,D6
        BEQ         FIN         si c'est égal, bye bye!
```

Ouf! Ce n'est pas la dernière lettre; nous pouvons donc afficher notre phrase. Cela se fait avec Gemdos 9, en lui passant l'adresse du début de la phrase dans la pile. Cette adresse c'est TXT et le Gemdos affichera jusqu'à ce qu'il rencontre 0. Il affichera donc 27,"Y",42,43,255,0. Ceci étant fait, comparons la lettre que nous venons d'afficher, et qui se trouve en face de l'étiquette LETTRE avec celle qui se trouve dans D6 et qui a été prélevée dans la phrase modèle.

Si c'est la même, nous remontons jusqu'à l'étiquette PROCHAINE, nous changeons de colonne, nous prélevons la lettre suivante dans la phrase modèle et nous recommençons. Mais si ce n'est pas la même lettre?

Et bien nous diminuons de 1 le code ASCII de 'LETTRE' (SUB.B #1,LETTRE) et nous ré-affichons notre phrase qui est maintenant 27,"Y",42,43,254,0

C'est compris ?

La aussi c'est une bonne étude qui vous permettra de vous en sortir.

N'abandonner pas ce listing en disant "oh ça va j'ai à peu près compris"
il faut PARFAITEMENT COMPRENDRE. N'hésitez pas à vous servir de MONST pour aller voir à l'adresse de LETTRE ce qui s'y passe. Pour avoir les adresses des étiquettes, taper L quand vous êtes sous MONST. Il est tout à fait possible de demander à ce que la fenêtre mémoire (la 3) pointe sur une partie vous montrant LETTRE et COLONE, puis de revenir sur la fenêtre 2 pour faire avancer pas à pas le programme. Ceci vous permettra de voir le contenu de la mémoire se modifier tout en regardant les instructions s'exécuter.

Il reste un petit point à éclaircir, concernant le mot EVEN qui est situé dans la section data. Nous avons déjà compris (du moins j'espère) que l'assembleur ne faisait que traduire en chiffres des instructions, afin que ces ordres soient compris par la machine. Nous avons vu également que le 68000 n'aimait pas les adresses impaires (du moins nous ne l'avons pas encore vu, et ce n'est pas plus mal...). Lorsque l'assembleur traduit en chiffre les mnémoniques, il n'y a pas de souci à se faire, celles-ci sont toujours traduites en un nombre pair d'octets.

Malheureusement ce n'est pas forcément le cas avec les datas. En l'occurrence ici, le label CLS commence à une adresse paire (car avant lui il n'y a que des mnémoniques) mais à l'adresse CLS on ne trouve que 3 octets. Nous en déduisons que le label TXT va se trouver à une adresse impaire. Pour éviter cela, l'assembleur met à notre disposition une instruction qui permet d'imposer une

adresse paire pour le label suivant, EVEN signifiant pair en Anglais.

Note: Tout comme SECTION DATA, DC.B, DC.W ou DC.L, EVEN n'est pas une instruction du 68000. C'est un ordre qui sera compris par l'assembleur.

Généralement ces ordres sont compris par beaucoup d'assembleurs mais il existe parfois des variantes. Ainsi certains assembleurs demandent à avoir .DATA ou bien DATA et non pas SECTION DATA. De même pour certains assembleurs, les labels (étiquettes) doivent être impérativement suivis de 2 points. Il faut chercher dans la doc de son assembleur et faire avec, c'est la seule solution!
Notez cependant que ceci ne change en rien les mnémoniques!

Passons maintenant au dernier listing de ce cours, le numéro 2.

Ce listing affiche une image Degas dont le nom est inscrit en section data, à l'étiquette NOM_FICHIER. Il est bien évident que ce nom ne doit pas contenir de c cédille mais plutôt une barre oblique inversée, que mon imprimante a refusée d'imprimer!

Seules 2 ou 3 petites choses vous sont inconnues. Tout d'abord l'instruction TST.W (juste après l'ouverture du fichier image)
Cette instruction se lit Test et donc ici on lit:
Test word D0.

Cela revient tout simplement à faire CMP.W #0,D0.

Seconde chose qui vous est encore inconnue, la SECTION BSS.

Nous avons vu dans les précédents que les variables initialisées étaient mises dans une SECTION DATA. Et bien les variables non initialisées sont mises dans une section nommée SECTION BSS. Cette section possède une particularité intéressante: les données y figurant ne prennent pas de place sur disque !

Ainsi si vous avez un programme de 3 kiloctets mais que dans ce programme vous désirez réserver 30 kilo pour pouvoir par la suite y charger différentes choses, si vous réservez en faisant TRUC DC.B 30000 votre programme, une fois sur disquette fera 33000 octets. Par contre si vous réservez par TRUC DS.B 30000, votre programme n'occupera que 3 Ko sur le disque.

Ces directives placées en section BSS sont assez différentes de celles placés en section data.

```
TRUC   DC.W     16   réserve de la place pour 1 word qui est
                         initialisé avec la valeur 16.
TRUC   DS.W     16   réserve de la place pour 16 words.
```

Il faut bien faire attention à cela, car c'est une faute d'étourderie peu fréquente mais ça arrive!
Si on note en section BSS

```
TRUC      DS.W       0
MACHIN    DS.W       3
```

Lorsque l'on cherchera le label TRUC et que l'on écrira des données dedans, ces données ne pourront pas aller DANS truc puisque cette étiquette ne correspond à rien (0 word de réservé) et donc nous écrirons dans MACHIN, en écrasant par exemple ce que nous y avions placé auparavant.

Bon, normalement vous devez en savoir assez long pour utiliser le Gemdos, le Bios et le Xbios (je vous rappelle que le Bios s'appelle par le Trap #13, exactement de la même manière que le Gemdos ou le Xbios).

Vous devez donc être capable de réaliser les programmes suivants:

Demande du nom d'une image. On tape le nom au clavier, puis le programme lit l'image sur la disquette et l'affiche. Prévient et redemande un autre nom si l'image n'est pas trouvée. Si on tape X, c'est la fin et on quitte le programme.

Lecture du premier secteur de la première piste de la disquette.
Si le premier octet de ce secteur est égale à $61 (c'est le code de l'instruction BRA), faire cling cling cling en affichant le code ASCII 7 (clochette), afficher "disquette infectée", attendre un appui sur une touche et bye bye. Si disquette non infectée, afficher "je remercie le Féroce Lapin pour ses excellents cours d'assembleur, super bien faits à que d'abord c'est lui le meilleur" et quitter.

Vous pouvez aussi tenter la vaccination, en effaçant carrément le premier octet (mettre à 0 par exemple).

Autre exemple assez intéressant à programmer. Vous avez vu dans le listing 3 comment prélever des données situées les unes après les autres dans une chaîne: D6 contient bien d'abord F puis E puis R etc... Imaginez que vous ayez 3 chaînes: la première contient des chiffres correspondant à la colonne d'affichage, la seconde des chiffres correspondant à la ligne et la troisième des chiffres correspondant à la couleurs, ces 3 données au format VT52.
(regardez Escape+'Y' et Escape+'b' ou Escape+'c'). On met un registre d'adresse pour chacune de ces listes, on lit un chiffre de chaque, on place ce chiffre dans une phrase:

```
 (27,"Y",X1,X2,27,"b",X3,"*",0)
```

X1 étant le chiffre prélevé dans la liste 1
X2 étant le chiffre prélevé dans la liste 2
X3 étant le chiffre prélevé dans la liste 3

On affiche donc à différentes positions une étoile, de couleur différente suivant les affichages.

Conseil: Essayez de faire le maximum de petits programmes, afin de bien comprendre l'utilisation du VT52, du Gemdos, du Bios et du Xbios. Cela vous permettra également de vous habituer à commenter vos programmes, à les ordonner, à chasser l'erreur sournoise.

Scrutez attentivement vos programmes à l'aide de MONST. Pour le moment les erreurs seront encore très faciles à trouver, il est donc impératif de très très bien vous entraîner!!!

Si un de vos programmes ne tourne pas, prenez votre temps et réfléchissez. C'est souvent une erreur ENORME qui est juste devant vous: notez sur papier les valeurs des registres, faites avancer pas à pas le programme sous MONST, repensez bien au principe de la pile avec ses avantages mais aussi ses inconvénients. Utilisez le principe des subroutines en y passant des paramètres afin de très bien maîtriser ce principe.

Vous recevrez la seconde série de cours dans un mois environ. Cela vous laisse le temps de bosser. Surtout approfondissez, et résistez à la tentation de désassembler des programmes pour essayez d'y comprendre quelque chose, ou à la tentation de prendre de gros sources en croyant y trouver des choses fantastiques. Ce n'est pas du tout la bonne solution, au contraire!!!

Si vraiment vous voulez faire tout de suite un gros trucs, alors faite un traitement de texte. Avec le VT52, le Gemdos et le Bios, c'est tout à fait possible. Bien sûr, il n'y aura pas la souris et il faudra taper le nom du fichier au lieu de cliquer dans le sélecteur, mais imaginez la tête de votre voisin qui frime avec son scrolling en comprenant 1 instruction sur 50 quand vous lui annoncerez "Le scrolling c'est pour les petits... moi je fais un traitement de texte!! "

De tout coeur, bon courage Le Féroce Lapin (from 44E)

Sommaire provisoire de la série 2

Reprogrammer les Traps,
Désassemblage et commentaire d'un programme dont nous ne sommes pas les auteurs,
la mémoire écran
les animations (scrolling verticaux, horizontaux, sprites, ...),
la musique (avec et sans digits,
les sound trackers...),
création de routines n'utilisant pas le système d'exploitation,
le GEM et les ressources etc....

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*                 COURS SUPLEMENTAIRE  réf. A                    *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                                                                *
******************************************************************
```

Ce chapitre a été rajouté alors que j'étais en train de rédiger le 6ème cours. Il m'a semblé, en effet, intéressant de vous fournir des détails supplémentaires sur le 68000.

Ces informations concernent le brochage de ce micro-processeur et peuvent sembler superflues. Elles permettent cependant une bien meilleure compréhension des divers phénomènes. Ne vous inquiétez pas si certains termes vous paraissent difficilement compréhensibles car nous ferons assez souvent appel à ce document dans les cours suivants, ce qui nous permettra d'obtenir des explications au fur et à mesure des besoins.

Le 68000 est composé d'une toute petite 'plaque' de silicium, à laquelle sont connectés des fils eux-mêmes terminés par de petites broches (les 'pattes'). C'est le nombre de broches qui conditionne la taille du boîtier et non la taille de la pastille de silicium, beaucoup plus petite.

Le boîtier du 68000 fait environ 8,2 cm de long sur 2,3 cm de large, et comporte 64 broches que nous allons décrire sommairement. Par simple souci pédagogique, l'ordre d'explication ne suit pas l'ordre numérique.

Pour chaque broche, vous trouverez son nom tel qu'il est donné dans les ouvrages sur le 68000, ce même nom en clair puis la description de la broche.

VCC Voltage constant current. Voltage en courant continue. C'est la broche d'alimentation du 68000 (5 volts)
GND ground. C'est la masse du 68000.

CLK Clock/Horloge. Entrée du signal d'horloge.

Note: On appel BUS un ensemble de conducteurs (en quelques sorte un 'paquet' de fils), véhiculant le même type d'information.

A1 à A23 Address / Adresse. Ces broches constituent le bus d'adresse. Il ne faut pas confondre ces 'A' avec les registres d'adresses A0-A7) que nous étudions dans les autres cours. En effet, chacun des 'A' des registres d'adresses est codé sur 32 bits alors qu'ici chacun ne travaille que sur 1 bit. Nous sommes bien ici en présence d'une boite avec des fils sur lesquels il y a ou non du courant (revoir cours 2). On pourrait s'attendre à trouver une broche 0, mais celle-ci est remplacée par 2 broches complétant le bus d'adresse.

UDS Upper Data Strobe/Echantillonnage haut
LDS Lower Data Strobe/Echantillonnage bas

A l'aide des broches A1-A23 on obtient une adresse, tandis que les broches UDS et LDS indique au micro-processeur si à cette adresse il doit accéder à l'octet haut, à l'octet bas ou au word complet.
Chaque broche A1-A23 ne pouvant prendre que 2 valeurs (0 ou 1) nous nous retrouvons dans le même cas que nos lampes du cours 2. Nous avions remarqué que le nombre de possibilité était lié au nombre de lampes par la relation:

possibilité = 2 à la puissance nombre de lampe.

Si nous remplaçons lampe par broche, nous obtenons comme nombre de possibilités 2 puissance 23, c'est à dire 8388608. Nous pouvons donc avoir accès à 8388608 adresses, chacune contenant non pas un octet mais un word, puisque le micro-processeur opère ensuite la sélection en consultant ses broches UDS et LDS. Nous pouvons donc atteindre 8388608 words c'est à dire 16777216 octets, ce qui fait bien les 16 mégas dont nous parlons dans les autres cours.

Le Bus de Données: Même remarques que précédemment. Ici nous avons 16 broches (D0 à D15) qui, bien sûr, ne peuvent prendre que 2 valeurs, à savoir 0 ou 1. Le bus de données est donc sur 16 bits, il est donc capable de véhiculer des bytes (octets) ou des words (mots). Il est possible de lire mais aussi d'écrire sur ce bus. Il est donc accessible dans les deux sens, on dit qu'il est bi-directionnel.

Le Bus de Control: Cet ensemble de broches fournit des informations complémentaires.

AS Adresse Strobe/Echantillonnage d'adresse. Cette broche

valide l'adresse se trouvant sur le bus d'adresse. En clair elle indique que tout est OK.

R/W Read-write/Lire-écrire. Le bus de donnée étant accessible en lecture et en écriture, cette broche indique lequel des 2 états est actif.

UDS,LDS Ces 2 broches font parties du Bus de Control mais nous les avons déjà décrites un peu plus haut.

DTACK Data Transfert Acknowledge / Récépissé de transfert de données.Indique que le transfert des données est réalisé.

Le Bus de Control comporte également d'autres commandes permettant une bonne répartition des bus suivant la demande.

BR Bus Request/demande de bus. Indique qu'un autre circuit demande à se rendre maître du bus.

BG Bus Grant Signale que le bus va être libéré.

BGACK Bus Grant Acknowledge. Indique qu'un autre circuit a pris la commande du bus.

Commande d'interruptions:

IPL Interrupt Pending Level: 3 broches de ce type IPL0, IPL1 et IPL2.

Ces broches, contrairement aux autres, sont actives lorsqu'elles sont à 0. Nous retrouverons plus tard l'état de ces broches dans le chapitre traitant de la seconde partie du SR et des interrupt (chapitre 4)

Commande du système.

BERR Bus error/Erreur de bus. Signale une erreur dans le cycle en cours d'exécution.

RESET Sert à initialiser le 68000. Cependant, lorsqu'un programme exécute l'instruction RESET, cette broche peut passer à l'état bas (0), afin qu'il y ait ré-initialisation des circuits externes sans toucher au 68000.

HALT Tout comme la broche RESET, celle-ci est disponible en entrée ou en sortie. Lorsqu'on l'attaque en entrée, le 68000 termine son cycle de bus en cours puis se bloque. En sortie cette broche indique une double faute intervenue sur un bus. Seul RESET peut alors débloquer le processeur.

Etat du processeur.

3 broches (FC2,FC1 et FC0) indique dans quel état se trouve le 68000.

FC2	FC1	FC0	Type de cycle
0	0	0	réservé (non utilisé)
0	0	1	données utilisateurs
0	1	0	programme utilisateur

```
0   1    1    réservé
1   0    0    réservé
1   0    1    données superviseur
1   1    0    programme superviseur
1   1    1    reconnaissance d'interruption
```

Vous trouverez de nombreuses autres informations sur le 68000 dans les ouvrages tels que "Mise en oeuvre du 68000" aux éditions Sybex, ou dans les ouvrages parus aux éditions Radio. Ces informations, même si elles ne paraissent pas primordiales, permettent de mieux comprendre le mode de fonctionnement de la machine, ce qui ne peut apporter que des avantages.

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*               Listing numéro 1 / Cours numéro 7                *
*                                                                *
******************************************************************

DEBUT          MOVE.L    #MENU,-(SP)    passe adresse phrase
               MOVE.W    #9,-(SP)       numéro fonction
               TRAP      #1             appel Gemdos
               ADDQ.L    #6,SP          correction pile

* attente appui touche
               MOVE.W    #7,-(SP)       fonction numéro 7
               TRAP      #1             du gemdos
               ADDQ.L    #2,SP          correction pile

* test du résultat
               CMP.W     #"Q",D0        q majuscule ?
               BEQ       QUITTER        oui, bye bye
               CMP.W     #"q",D0        q minuscule ?
               BEQ       QUITTER        oui, bye bye
               CMP.W     #"V",D0        v majuscule ?
               BEQ       AFFICHAGE      oui -> affiche le message
               CMP.W     #"v",D0        V minuscule ?
               BNE       DEBUT          non. On a donc une autre lettre
* comme réponse. Ce n'est pas valable donc on recommence au début

AFFICHAGE      MOVE.L    #MESSAGE,-(SP) adresse de 'coucou'
               MOVE.W    #9,-(SP)       numéro fonction
               TRAP      #1             appel Gemdos
               ADDQ.L    #6,SP          correction pile
* On attend un appui sur une touche pour contempler
               MOVE.W    #7,-(SP)
               TRAP      #1
               ADDQ.L    #2,SP
```

```
               BRA       DEBUT          retour au début

QUITTER        MOVE.W    #0,-(SP)
               TRAP      #1

    SECTION DATA

MENU           DC.B 27,"E",27,"p"
               DC.B "QUITTER (Q) OU VOIR LE MESSAGE (V) ?"
               DC.B 27,"q",13,10,0
               EVEN
MESSAGE        DC.B "COUCOU",0
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
  **********************************************************************
**
  *
*
  *                COURS D'ASSEMBLEUR 68000 SUR ATARI ST
*
  *
*
  *                    par Le Féroce Lapin (from 44E)
*
  *
*
  *                  Listing numéro 2 / Cours numéro 7
*
  *
*
  **********************************************************************
**

  *  Test résolution, car si on est en haute et que l'on essaye de *
  *  passer en basse, paf!  RESET !                                *

         MOVE.W    #4,-(SP)                numéro fonction  Getrez()
         TRAP      #14                     appel Xbios
         ADDQ.L    #2,SP                   correction pile
         CMP.W     #2,D0                   haute ?
         BEQ       CHARGE                  oui, donc on charge tout de
                                           suite

  * Puisqu'on est pas en haute, on force en basse résolution
  * on note d'abord en mettant à 1 un drapeau de signalement

         MOVE.W    #1,RESOLUTION

         MOVE.W    #0,-(SP)                basse résolution
         MOVE.L    #-1,-(SP)               adresse écran inchangée
         MOVE.L    #-1,-(SP)               adresse écran inchangée
         MOVE.W    #5,-(SP)                fonction Setscreen()
         TRAP      #14                     du X bios
         ADDA.L    #12,SP                  correction
```

```
* Ouverture du fichier image

CHARGE   MOVE.W    #0,-(SP)              ouverture en lecture
         MOVE.0L   #NOM_FICHIER,-(SP)    adr nom du fichier
         MOVE.W    #61,-(SP)             fonction Fopen()
         TRAP      #1                    du GEMDOS
         ADDQ.L    #8,SP                 correction pile


* D0 contient le Handle ou bien un numéro d'erreur négatif

         TST.W     D0                    compare à 0
         BLT       ERREUR                inférieur donc erreur


* On sauve le handle du fichier

         MOVE.W    D0,D3


* Saute les 34 octets du début de fichier DEGAS
* (2 octets d'en-téte, 32 de couleurs)

         MOVE.W    #0,-(SP)              décale à partir du début de
                                         fichier
         MOVE.W    D3,-(SP)              handle du fichier
         MOVE.L    #34,-(SP)             nbr d'octets à sauter
         MOVE.W    #66,-(SP)             fonction Fseek()
         TRAP      #1                    du GEMDOS
         ADDA.L    #10,SP
         TST.W     D0                    test D0
         BLT       ERREUR


* Détermine l'adresse de l'écran

         MOVE.W    #2,-(SP)              fonction Physbase()
         TRAP      #14                   du xbios
         ADDQ.L    #2,SP                 correction pile
         MOVE.L    D0,A5                 sauve l'adresse


* Charge l'image directement dans l'écran

         MOVE.L    A5,-(SP)              adresse destination
         MOVE.L    #32000,-(SP)          nbr octets à lire
         MOVE.W    D3,-(SP)              handle du fichier
         MOVE.W    #63,-(SP)             fonction Fread()
         TRAP      #1                    du GEMDOS
         ADDA.L    #12,SP
         TST.W     D0
         BLT       ERREUR


* Chargement de la palette dans notre buffer palette
* D'abord repositionner le pointeur fichier

         MOVE.W    #0,-(SP)              décale à partir du début de
                                         fichier
         MOVE.W    D3,-(SP)              handle du fichier
         MOVE.L    #2,-(SP)              nbr d'octets à sauter
```

```
         MOVE.W    #66,-(SP)              fonction Fseek()
         TRAP      #1                     du GEMDOS
         ADDA.L    #10,SP
         TST.W     D0                     test D0
         BLT       ERREUR


* Puis chargement

         MOVE.L    #BUFFER_PAL,-(SP)      adresse destination
         MOVE.L    #32,-(SP)              nbr octets à lire
         MOVE.W    D3,-(SP)               handle du fichier
         MOVE.W    #63,-(SP)              fonction Fread()
         TRAP      #1                     du GEMDOS
         ADDA.L    #12,SP
         TST.W     D0
         BLT       ERREUR


* On place maintenant cette palette avec Xbios (6)

         MOVE.L    #BUFFER_PAL,-(SP)      adresse palette noire
         MOVE.W    #6,-(SP)               fonction SetPalette()
         TRAP      #14                    du XBIOS
         ADDQ.L    #6,SP


* On referme le fichier

         MOVE.W    D3,-(SP)               handle
         MOVE.W    #62,-(SP)              fonction Fclose()
         TRAP      #1                     du GEMDOS
         ADDQ.L    #4,SP

         BRA       FIN                    et on se sauve


* En cas d'erreur on vient ici

ERREUR   MOVE.L    #MESSAGE_ERREUR,-(SP)  prévient
         MOVE.W    #9,-(SP)
         TRAP      #1
         ADDQ.L    #6,SP

FIN      MOVE.W    #7,-(SP)               attend un appui touche
         TRAP      #1
         ADDQ.L    #2,SP

         CMP.W     #0,RESOLUTION
         BEQ       NOT_MOYENNE


* Puisqu'on est en basse, on reforce en moyenne résolution

         MOVE.W    #1,-(SP)               moyenne résolution
         MOVE.L    #-1,-(SP)              adresse écran inchangée
         MOVE.L    #-1,-(SP)              adresse écran inchangée
         MOVE.W    #5,-(SP)               fonction Setscreen()
         TRAP      #14                    du XBIOS
         ADDA.L    #12,SP                 correction


* Puis on quitte
```

```
NOT_MOYENNE
          MOVE.W     #0,-(SP)
          TRAP       #1
*--------------------------------------------------------------*
    SECTION DATA
MESSAGE_ERREUR     DC.B                    "Désolé, erreur avec",13,10
                   DC.B                    "le fichier "
NOM_FICHIER        DC.B                    "A:\IMAGE.PI1",0

    SECTION BSS
BUFFER_PAL         DS.W                    16
RESOLUTION         DS.W                    1
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
******************************************************************
*                                                                *
*           COURS D'ASSEMBLEUR 68000 SUR ATARI ST                *
*                                                                *
*               par Le Féroce Lapin (from 44E)                   *
*                                                                *
*             Listing numéro 3 / Cours numéro 7                  *
*                                                                *
******************************************************************


                 ######## CRC ERROR ########
```

Le fichier d'où est tiré ce listing était corrompu sur mes vieilles disquettes, je n'ai donc pas pu le convertir. Si vous l'avez en entier, merci de me l'envoyer.

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
SHIZUKA a le plaisir de vous présenter cette table des cycles du 68000
réalisée en deux portions de soirées avec les aides appréciables de:

craft2 ( stedi )
disque dur ( ouaaaah la vitesse )
"guide des instructions du 68000 , vol 2 de NACHTMANN chez PUBLITRONIC
"

merci de diffuser cette table sans modifier l'en tête

calcul de l'adresse effective

                b/w     l
```

dn	0	0
an	0	0
(an)	4	8
(an)+	4	8
-(an)	6	10
d16(an)	8	12
d8(an,xi)	10	14
abs.w	8	12
abs.l	12	16
d16(pc)	8	12
d8(pc,x1)	10	14
imm	4	8

move.b et move.w

	dn	an	(an)	(an)+	-(an)	d16(an)	d8(an,xi)	abs.w	abs.l
dn	4	4	8	8	8	12	14	12	16
an	4	4	8	8	8	12	14	12	16
(an)	8	8	12	12	12	16	18	16	20
(an)+	8	8	12	12	12	16	18	16	20
-(an)	10	10	14	14	14	18	20	18	22
d16(an)	12	12	16	16	16	20	22	20	24
d8(an,xi)	14	14	18	18	18	22	24	22	26
abs.w	12	12	16	16	16	20	22	20	24
abs.l	16	16	20	20	20	24	26	24	28
d16(pc)	12	12	16	16	16	20	22	20	24
d8(pc,x1)	14	14	18	18	18	22	24	22	26
imm	8	8	12	12	12	16	18	16	20

move.l

	dn	an	(an)	(an)+	-(an)	d16(an)	d8(an,xi)	abs.w	abs.l
dn	4	4	12	12	12	16	18	16	20
an	4	4	12	12	12	16	18	16	20
(an)	12	12	20	20	20	24	26	24	28
(an)+	12	12	20	20	20	24	26	24	28
-(an)	14	14	22	22	22	26	28	26	30
d16(an)	16	16	24	24	24	28	30	28	32
d8(an,xi)	18	18	26	26	26	30	32	30	34
abs.w	16	16	24	24	24	28	30	28	

32

abs.l	20	20	28	28	28	32	34	32	36
d16(pc)	16	16	24	24	24	28	30	28	32
d8(pc,x1)	18	18	26	26	26	30	32	30	34
imm	12	12	20	20	20	24	26	24	28

	op(ea),an	op(ea),dn	op dn,(m)
add.b	-	4+	8+
add.w	8+	4+	8+
add.l	6+&	6+&	12+
and.b	-	4+	8+
and.w	-	4+	8+
and.l	-	6+&	12+
cmp.b	-	4+	-
cmp.w	6+	4+	-
cmp.l	6+	6+	-
divs	-	158+*	-
divu	-	140+*	-
eor.b	-	4+	8+
eor.w	-	4+	8+
eor.l	-	8+	12+
muls	-	70+*	-
mulu	-	70+*	-
or.b	-	4+	8+
or.w	-	4+	8+
or.l	-	6+&	12+
sub.b	-	4+	8+
sub.w	8+	4+	8+
sub.l	6+&	6+&	12+

+ = rajouter le temps de calcul de l'adresse effective
& = raouter 2 périodes d horloge pour les modes an dn et immédiat
* = durée max

	op #,dn	op #,an	op #,m
addi.b	8	-	12+
addi.w	8	-	12+
addi.l	16	-	20+
addq.b	4	-	8+
addq.w	4	8	8+
addq.l	8	8	12+
andi.b	8	-	12+
andi.w	8	-	12+
andi.l	16	-	20+
cmpi.b	8	-	8+
cmpi.w	8	-	8+
cmpi.l	14	-	12+
eori.b	8	-	12+
eori.w	8	-	12+
eori.l	16	-	20+
moveq	4	-	-
ori.b	8	-	12+

ori.w	8	-	12+
ori.l	16	-	20+
subi.b	8	-	12+
subi.w	8	-	12+
subi.l	16	-	20+
subq.b	4	-	8+
subq.w	4	8	8+
subq.l	8	8	12+

+ = rajouter le temps de calcul de l'adresse effective

	registre	mémoire
clr.b	4	8+
clr.w	4	8+
clr.l	6	12+
nbcd	6	8+
neg.b	4	8+
neg.w	4	8+
neg.l	6	12+
negx.b	4	8+
negx.w	4	8+
negx.l	6	12+
not.b	4	8+
not.w	4	8+
not.l	6	12+
scc (cc=0)	4	8+
scc (cc=1)	6	8+
tas	4	10+
tst.b	4	4+
tst.w	4	4+
tst.l	4	4+

+ = rajouter le temps de calcul de l'adresse effective

	registres	memoire
asr.b & asl.b	6+2n	
asr.w & asl.w	6+2n	8+
asr.l & asl.l	8+2n	
lsr.b & lsl.b	6+2n	
lsr.w & lsl.w	6+2n	8+
lsr.l & lsl.l	8+2n	
ror.b & rol.b	6+2n	
ror.w & rol.w	6+2n	8+
ror.l & rol.l	8+2n	
roxr.b & roxl.b	6+2n	
roxr.w & roxl.w	6+2n	8+
roxr.l & roxl.l	8+2n	

+ = rajouter le temps de calcul de l'adresse effective
n est le nombre de rotations ou de décalages successifs

	dynamique registre	dynamique mmemoire	statique registre	statique memoire
bchg.b	-	8+	-	12+
bchg.l	8*	-	12*	-
bclr.b	-	8+	-	12+
bclr.l	10*	-	14*	-

bset.b	-	8+	-	12+
bset.l	8*	-	12*	-
btst.b	-	4+	-	8+
btst.l	6	-	10	-

+ = rajouter le temps de calcul de l'adresse effective
* = durée max

	branchement effectué	pas de branchement
bcc.s	10	8
bcc.l	10	12
bra.s	10	-
bra.l	10	-
bsr.s	18	-
bsr.l	18	-
dbcc cc=1	-	12
dbcc cc=0	10	14

	(an)	(an)+	-(an)	16(an)	d8(an,xi)	abs.w	abs.l	d16(pc)	d8(pc,xi)
jmp	8	-	-	10	14	10	12	10	14
jsr	16	-	-	18	22	18	20	18	22
lea	4	-	-	8	12	8	12	8	12
pea	12	-	-	16	20	16	20	16	20
movem.w (m->r)	12+4n	12+4n	-	16+4n	18+4n	16+4n	20+4n	16+4n	18+4n
movem.l (m->r)	12+8n	12+8n	-	16+8n	18+8n	16+8n	20+8n	16+8n	18+8n
movem.w (r->m)	8+4n	-	8+4n	12+4n	14+4n	12+4n	16+4n	-	-
movem.l (r->m)	8+8n	-	8+8n	12+8n	14+8n	12+8n	16+8n	-	-

n est le nombre de transferts de registra à effectuer

	op dn,dn	op m,m
addx.b	4	18
addx.w	4	18
addx.l	8	30
cmpm.b	-	12
cmpm.w	-	12
cmpm.l	-	20
subx.b	4	18
subx.w	4	18

subx.l	8	30
abcd	6	18
sbcd	6	18

	68000 r->m	68000 m->r
movep.w	16	16
movep.l	24	24

	68000 registre	68000 mémoire
andi to ccr	20	-
andi to sr	20	-
chk (sans trap)	10	-
eori to ccr	20	-
eori to sr	20	-
exg	6	-
ext	4	-
link	16	-
move to ccr	12	12&
move to sr	12	12&
move from sr	6	8&
move usp	4	-
nop	4	-
ori to ccr	20	-
ori to sr	20	-
reset	132	-
rte	20	-
rtr	20	-
rts	16	-
stop	4	-
swap	4	-
trapv (sans trap)	4	-
unlk	12	-

& = rajouter le temps de calcul de l'adresse effective

	68000
erreur adresse	50
erreur de bus	50
instruction chk	44+*
instruction illegal	34
interruption	44#
violation de privilège	34
trace	34
trap	38
trapvs	34

+ rajouter l temps de calcul de l'adresse effective
* durée max
4 périodes d'horloge incluses pour le cycle d'interruption

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
***-------------- LES FONCTIONS GEMDOS PAR NUMÉRO ---------------***

$00      Pterm0    Fin de processus
$01      Cconin    Lit un caractère sur l'entrée standard
$02      Cconout   Écrit un caractère sur la sortie standard
$03      Cauxin    Lit un caractère sur l'entrée standard AUX:
$04      Cauxout   Écrit un caractère sur la sortie standard AUX:
$05      Cprnout   Écrit un caractère sur la sortie standard PRN:
$06      Crawio    Lecture/écriture brute sur l'entrée/sortie standard
$07      Crawcin   Lecture brute sur l'entrée standard
$08      Cnecin    Lit un caractère sur l'entrée standard, sans écho
$09      Cconws    Écrit une chaîne sur la sortie standard
$0A      Cconrs    Lit une chaîne formatée sur l'entrée standard
$0B      Cconis    Teste l'état de l'entrée standard
$0E      Dsetdrv   Fixe le lecteur de disque par défaut
$10      Cconos    Teste l'état de la sortie standard
$11      Cprnos    Teste l'état du périphérique standard PRN:
$12      Cauxis    Teste l'état du standard AUX: en entrée
$13      Cauxos    Teste l'état du standard AUX: en sortie
$19      Dgetdrv   Demande le disque par défaut
$1A      Fsetdta   Fixe l'adresse du DTA (Disk Transfer Adress)
$20      Super     Entre/Sort/Demande du mode superviseur
$2A      Tgetdate  Demande la date
$2B      Tsetdate  Fixe la date
$2C      Tgettime  Demande l'heure
$2D      Tsettime  Fixe l'heure
$2F      Fgetdta   Demande l'adresse du DTA (Disk Transfer Adress)
$30      Sversion  Demande le numéro de version du GEMDOS
$31      Ptermres  Finit un programme qui reste résident
$36      Dfree     Demande d'informations sur un disque
$39      Dcreate   Création d'un sous-répertoire
$3A      Ddelete   Efface un sous-répertoire
$3B      Dsetpath  Fixe le catalogue courant
$3C      Fcreate   Création d'un fichier
$3D      Fopen     Ouvre un fichier
$3E      Fclose    Ferme un fichier
$3F      Fread     Lit un fichier
$40      Fwrite    Écrit un fichier
$41      Fdelete   Efface un fichier
$42      Fseek     Positionnement dans un fichier
$43      Fattrib   Retourne/fixe les attributs de fichier
$45      Fdup      Recopie un identificateur de fichier standard
$46      Fforce    Force un identificateur de fichier
$47      Dgetpath  Demande le répertoire courant
$48      Malloc    Demande d'allocation mémoire
$49      Mfree     Libère de la mémoire
$4A      Mshrink   Rétrécit un bloc de mémoire allouée
$4B      Pexec     Charge/Exécute un programme
$4C      Pterm     Termine un programme
$4E      Fsfirst   Recherche la première occurrence d'un fichier
$4F      Fnext     Recherche l'occurrence suivante
$56      Frename   Renomme un fichier
$57      Fdatime   Demande ou fixe la date de création d'un fichier
```

```
***-------------- FONCTIONS GEMDOS PAR NOM ----------------------***

$03      Cauxin    Lit un caractère sur l'entrée standard AUX:
$12      Cauxis    Teste l'état du standard AUX: en entrée
$13      Cauxos    Teste l'état du standard AUX: en sortie
$04      Cauxout   Écrit un caractère sur la sortie standard AUX:
$01      Cconin    Lit un caractère sur l'entrée standard
$0B      Cconis    Teste l'état de l'entrée standard
$10      Cconos    Teste l'état de la sortie standard
$02      Cconout   Écrit un caractère sur la sortie standard
$0A      Cconrs    Lit une chaîne formatée sur l'entrée standard
$09      Cconws    Écrit une chaîne sur la sortie standard
$08      Cnecin    Lit un caractère sur l'entrée standard, sans écho
$11      Cprnos    Teste l'état du périphérique standard PRN:
$05      Cprnout   Écrit un caractère sur la sortie standard PRN:
$07      Crawcin   Lecture brute sur l'entrée standard
$06      Crawio    Lecture/écriture brute sur l'entrée/sortie standard
$39      Dcreate   Création d'un sous-répertoire
$3A      Ddelete   Efface un sous-répertoire
$36      Dfree     Demande d'informations sur un disque
$19      Dgetdrv   Demande le disque par défaut
$47      Dgetpath  Demande le répertoire courant
$0E      Dsetdrv   Fixe le lecteur de disque par défaut
$3B      Dsetpath  Fixe le catalogue courant
$43      Fattrib   Retourne/fixe les attributs de fichier
$3E      Fclose    Ferme un fichier
$3C      Fcreate   Création d'un fichier
$57      Fdatime   Demande ou fixe la date de création d'un fichier
$41      Fdelete   Efface un fichier
$45      Fdup      Recopie un identificateur de fichier standard
$46      Fforce    Force un identificateur de fichier
$2F      Fgetdta   Demande l'adresse du DTA (Disk Transfer Adress)
$3D      Fopen     Ouvre un fichier
$3F      Fread     Lit un fichier
$56      Frename   Renomme un fichier
$42      Fseek     Positionnement dans un fichier
$1A      Fsetdta   Fixe l'adresse du DTA (Disk Transfer Adress)
$4E      Fsfirst   Recherche la première occurrence d'un fichier
$4F      Fnext     Recherche l'occurrence suivante
$40      Fwrite    Écrit dans un fichier
$48      Malloc    Demande d'allocation mémoire
$49      Mfree     Libère de la mémoire
$4A      Mshrink   Rétrécit un bloc de mémoire allouée
$4B      Pexec     Charge/Exécute un programme
$4C      Pterm     Termine un programme
$00      Pterm0    Termine un programme (code de retour 0)
$31      Ptermres  Termine un programme qui reste résident
$20      Super     Entre/Sort/Demande du mode superviseur
$30      Sversion  Demande le numéro de version du GEMDOS
$2A      Tgetdate  Demande la date
$2C      Tgettime  Demande l'heure
$2B      Tsetdate  Fixe la date
$2D      Tsettime  Fixe l'heure

*---------------------------------------------------------------*
$00      Pterm0    Finit un programme

     void Pterm0()
```

Termine un processus, fermant tous les fichiers qu'il a ouvert et libérant la mémoire qu'il a allouée. Retourne $0000 comme code de sortie au programme parent.

$01 Cconin   Lit un caractère sur l'entrée standard

LONG Cconin()

Lit un caractère sur l'entrée standard (identificateur 0). Si l'entrée standard est la console, le long_mot retourné dans D0 contient le code ASCII et le code clavier de la touche appuyée (code de scrutation) :

31..24	23.. 16	15..8	7..0
$00 ou bits de shift	code de scrutation ou $00	$00	caractère ASCII

Les touches de fonction (F1 à F10, HELP, UNDO, etc...) retournent le code ASCII $00, avec le code clavier approprié (cf. le manuel sur le clavier intelligent pour la valeur des codes clavier). Le ST BIOS place l'état des touches spéciales dans les bits 24 à 31 (voir le Guide du programmeur du BIOS pour plus de détails).

ERREURS

Ne retourne aucune indication de fin de fichier.
Ne reconnaît pas 'Control_C'
Impossible de savoir si l'entrée est un périphérique à caractères ou un fichier.
Il devrait exister un moyen de taper les 256 codes possibles à partir du clavier.

$02 Cconout   Ecrit un caractère sur la sortie standard

void Cconout(c)
WORD c

Écrit le caractère 'c' sur la sortie standard (identificateur 0). Les huit bits les plus significatifs de 'c' sont réservés et doivent être à zéro. Les tabulations ne sont pas interprétées.

$03 Cauxin   Lit un caractère sur l'entrée standard AUX:

WORD Cauxin()

Lit un caractère à partir de l'identificateur 1 (le port série AUX:, normalement).

REMARQUE

Le contrôle de flux de la sortie RS232 ne marche pas avec cette fonction. Les programmes devraient utiliser l'appel de périphérique de caractère BIOS afin d'éviter de perdre des caractères reçus.

$04 Cauxout   Écrit un caractère sur la sortie standard AUX:

void Cauxout(c)
WORD c

$05 Cprnout   Écrit un caractère sur la sortie standard PRN:

void Cprnout(c)
WORD c

Écrit 'c' sur l'identificateur 2 (le port imprimante PRN:, norma-

lement). Les huits bits hauts de 'c' sont réservés et doivent être à zéro. Les tabulations ne sont pas interprétées.

--

$06 Crawio   Lecture/Écriture brute sur l'entrée/sortie standard

```
     LONG Crawio(w)
     WORD w;
```

REMARQUES

Par le fait même de sa conception cette fonction ne peut écrire '$ff' sur la sortie standard.
$00 ne peut être différencié d'une fin de fichier.

--

$07 Crawcin   Entrée brute sur l'entrée standard

```
     LONG Crawcin()
```

REMARQUE   Pas d'indication de fin de fichier.

--

$08 Cnecin   Lit un caractère sur l'entrée standard, sans écho

```
     LONG Cnecin()
```

Lit un caractère sur l'entrée standard. Si le périphérique d'entrée est 'CON:', aucun écho n'est fait. Les caractères de contrôle sont interprétés.

--

$09 Cconws   Ecrit une chaîne sur la sortie standard

```
     void Cconws(chaîne)
     char *chaîne;
```

Écrit une chaîne débutant à 'chaîne' et finissant par 0, sur la sortie standard.

--

$0A Cconrs   Lit une chaîne sur l'entrée standard

```
     void Cconrs(buf)
     char *buf;
```

Lit une chaîne sur l'entrée standard. Les caractères de contrôle habituels sont interprétés :

Caractère	Fonction
, ^J	Fin de ligne
^H,	Efface le dernier caractère
^U, ^X	Efface la ligne entière
^R	Recopie la ligne
^C	Fini le programme

Le premier caractère de 'buf' contient le nombre maximum d'octets à lire (taille du tampon moins deux). En sortie, le deuxième octet contient le nombre de caractères lus. La chaîne se trouve entre 'buf+2' et 'buf+2+buf[1]'.
Il n'est pas garanti que la chaîne se finisse par 0.

REMARQUE   Plante sur les fins de fichiers.

--

$0B Cconis   Teste l'état de l'entrée standard

```
     WORD Cconis()
```

Retourne $FFFF si un caractère est disponible sur l'entrée standard, $0000 sinon.

--

$0E Dsetdrv   Fixe le lecteur de disque par défaut

```
     LONG Dsetdrv(drv)
     WORD drv;
```

Fixe le disque 'drv' par défaut. Les disques sont rangés de 0 à 15 (A: à P:). Retourne dans D0.L la carte des disques actifs:
(bit 0 = A, bit 1 = B, etc..).

Un 'disque actif' est un disque à partir duquel un catalogue a été fait.

REMARQUE Le GEMDOS ne supporte que 16 disques (bits 0 à 15). Les systèmes ultérieurs en supporteront 32 .

$10 Cconos   Teste l'état de la sortie standard
```
WORD Cconos()
```

Renvoie $FFFF si la console est prête à recevoir un caractère, $0000 si la console n'est PAS prête.

$11 Cprnos   Teste l'état de la sortie standard PRN:
```
WORD Cprnos()
```
Retourne $FFFF si 'PRN:' est prêt à recevoir un caractère, $0000 sinon.

$12 Cauxis   Teste l'état de l'entrée standard AUX:
```
WORD Cauxis()
```
Retourne $FFFF si un caractère est disponible sur l'entrée 'AUX:' (identificateur 1), $0000 sinon.

$13 Cauxos   Teste l'état de la sortie standard AUX:
```
WORD Cauxos()
```
Renvoie $FFFF si 'AUX:' (identificateur 1) est prêt à recevoir un caractère, $0000 sinon.

$19 Dgetdrv   Recherche le lecteur de disque par défaut
```
WORD Dgetdrv()
```
Retourne le numéro du lecteur courant, compris entre 0 et 15.

$1A Fsetdta   Fixe l'adresse du DTA (Disk Transfer Adress)
```
void Fsetdta(adr)
char adr;
```
Fixe l'adresse du DTA à 'adr'. Le tampon de stockage des données sur un fichier (DTA) ne sert que pour les fonctions Fsfirst() et Fsnext().

$20 Super   Change/teste le mode (Utilisateur ou Superviseur)
```
LONG Super(pile)
WORD *pile;
```
'pile' est égal à -1L ($FFFFFFFF):
la fonction retourne $0000 si le processeur est en mode Utilisateur, $0001 s'il est en mode Superviseur.

'pile' est différent de -1L:

si le processeur est en mode Utilisateur, la fonction revient avec le processeur en mode Superviseur. Si 'pile' est NUL ($00000000), la pile superviseur sera la même que la pile Utilisateur avant l'appel. Sinon la pile Superviseur sera placée à 'pile'.

Si le processeur était en mode Superviseur, il se retrouve en mode Utilisateur au retour de la fonction. 'pile' devra être la valeur de la pile utilisateur qui a été retournée par le premier appel de la fonction.

ATTENTION

La pile Superviseur originale doit être replacée avant la fin

du programme sinon le système plantera à la sortie du programme.

```
*------------------------------------------------------------------*
```

$2A Tgetdate Demande la date

WORD Tgetdate()

Retourne la date en format DOS :

```
15 __________________9_8_________5_4_________0
|  Année depuis 1980  |    mois    |   jour    |
|     0.....119       |  1...12    |  1...31   |
|_____________________|____________|___________|
```

RETOUR

les bits 0 à 4 contiennent le jour allant de 1 à 31.
les bits 5 à 8 contiennent le mois allant de 1 à 12.
les bits 9 à 15 contiennent l'année (à partir de 1980) allant de 0 à 119.

```
*------------------------------------------------------------------*
```

$2B Tsetdate Fixe la date

WORD Tsetdate(date)
WORD date;

Fixe 'date' comme date courante dans le format décrit dans Tgetdate().

RETOUR 0 si la date est valide.
ERROR si le format de la date est incorrect.

REMARQUES

Le GEMDOS n'est pas difficile sur la date; par exemple le 31 Février ne lui déplaît pas.
Le GEMDOS ne prévient PAS le BIOS que la date a changée.

```
*------------------------------------------------------------------*
```

$2C Tgettime Demande l'heure

WORD Tgettime()

Retourne l'heure courante dans le format DOS:

```
15__________11_10_________5_4___________0
|    heure      |   minute  |   seconde  |
|   0...23      |   0...59  |   0...28   |
|_______________|___________|____________|
```

RETOUR Les bits :

0 à 4 contiennent les secondes divisées par 2 (0 à 28)
5 à 10 contiennent les minutes (0 à 59)
11 à 15 contiennent les heures (0 à 23)

```
*------------------------------------------------------------------*
```

$2D Tsettime Fixe l'heure

WORD Tsettime(heure)
WORD heure;

Fixe 'heure' comme heure courante dans le format décrit dans Tgettime().

RETOUR 0 si le format de l'heure fournie est valide;
ERROR si le format de l'heure n'est pas valide.

REMARQUE

Le GEMDOS ne prévient pas le BIOS que l'heure a changé.

```
*------------------------------------------------------------------*
```

$2F Fgetdta Demande l'adresse du DTA (Disk Transfer Address)

LONG Fgetdta()

Retourne l'adresse courante du tampon de stockage des données sur un

fichier (DTA), employée par les fonctions Fsfirst() et Fsnext().
--
$30 Sversion Demande le numéro de version du GEMDOS
```
     WORD Sversion()
```

Retourne le numéro de version du GEMDOS en format inversé. L'octet le plus significatif contient la partie basse du numéro, l'octet le moins significatif, la partie haute.

REMARQUES

La version du GEMDOS sur disquette du 29/5/85 et la première version en ROM du 20/11/85 ont $1300 comme numéro.

Les numéros de version du GEMDOS et du TOS ne sont PAS les mêmes. (cf. LE MANUEL DE REFERENCE DU BIOS DU ST pour le numéro de version du TOS).
--
$31 Ptermres Finit un programme qui reste résident
```
     void Permres( nbrest,coderet)
     LONG nbrest;
     WORD coderet;
```

Finit le programme courant, en conservant une part de sa mémoire. 'nbrest' le est nombre d'octets appartenant au programme qui doivent être gardés, comprenant et commençant à la page de base. 'coderet' est le code de sortie qui est retourné au programme parent.

La mémoire que le programme a allouée (en plus de sa zone TPA) N'EST PAS libérée.
Il est impossible de rappeler le programme terminé par Ptermres().

REMARQUE

Les fichiers ouverts sont fermés lors de la fin de processus.
--
$36 Dfree   Demande l'espace libre sur un disque
```
     void Dfree(buf,nbdisq)
     LONG *buf;
     WORD nbdisq;
```

Demande des informations sur le disque 'nbdisq' et les place dans quatre longs_mots commençant à 'buf':

buf + 0	nombre de blocs libres sur le disque
buf + 4	nombre total de blocs sur le disque
buf + 8	taille d'un secteur en octets
buf + 12	nombre de secteurs par bloc

REMARQUE   Incroyablement lent sur un disque dur (5 à 10 secondes).
--
$39 Dcreate   Création d'un sous_répertoire de disque C
```
     WORD Dcreate(chemin)
     char *chemin;
```

'Chemin' pointe sur une chaîne terminée par 0 spécifiant le chemin du nouveau répertoire.

RETOUR

0 en cas de succès;
ERROR ou le numéro d'erreur approprié en cas d'échec.

--
$3A Ddelete   Efface un sous_répertoire

WORD Ddelete(chemin)
char *chemin;

Efface un répertoire qui doit être vide (il peut toutefois comporter les fichiers spéciaux "." et ".."). 'chemin' pointe sur une chaîne terminée par zéro, spécifiant le chemin du répertoire à effacer.

RETOUR   0 en cas de succès;
ERROR ou le numéro d'erreur approprié en cas d'échec.

--
$3B Dsetpath Fixe le répertoire courant

WORD Dsetpath(chemin)
char *chemin;

Fixe 'chemin', (une chaîne terminée par 0), comme répertoire courant. Si le chemin commence par une lettre identificatrice de disque suivie de deux-points, le répertoire courant est placé sur le lecteur spécifié.

Le système conserve un répertoire courant pour chaque lecteur.

RETOUR   0 en cas de succès;
ERROR ou le numéro d'erreur approprié en cas d'echec.

--
$3C Fcreate   Création d'un fichier

WORD Fcreate(nomfich,attribs)
char *nomfich;
WORD attribs;

Crée un fichier 'nomfich' et retourne son identificateur, non standard, d'écriture seule. Le mot d'attributs est stocké dans le répertoire; les bits d'attributs sont :

masque	description
$01	fichier créé en lecture seule
$02	fichier transparent au répertoire
$04	fichier système
$08	fichier contenant un nom de volume sur 11 bits

RETOUR

Un nombre positif, l'identificateur ou :
ERROR ou le numéro d'erreur approprié.

REMARQUES

Si le bit de lecture seule est positionné, un identificateur d'écriture seule est retourné, et, de plus, cet identificateur ne permet pas d'écrire.

Théoriquement, un seul numéro de volume est permis sur un répertoire racine. Le GEMDOS n'impose rien de tel, ce qui peut causer quelque confusion.

--
$3D Fopen   Ouverture d'un fichier

WORD Fopen(nmfich,mode)
char *nmfich;
WORD mode;

Ouvre le fichier 'nmfich' en mode 'mode' et retourne son identificateur non standard. Le mode d'ouverture peut être:

mode	description

```
                  | 0     | lecture seule        |
                  | 1     | écriture seule       |
                  | 2     | lecture et écriture |
                  |______|_____________________|
```

RETOUR
     Un nombre positif, l'identificateur
     Un numéro d'erreur négatif.

$3E Fclose   Fermeture d'un fichier

```
     WORD Fclose(idfich)
     WORD idfich;
```

Ferme le fichier dont l'identificateur est 'idfich'.

RETOUR   0 en cas de succès;
         ERROR ou un numéro d'erreur approprié en cas d'échec.

$3F Fread   Lecture sur un fichier

```
     LONG Fread(idfich,nbre,buffer)
     WORD idfich;
     LONG nbre;
     char *buffer;
```

Lit 'nbre' octets dans le fichier d'identificateur 'idfich', et les place en mémoire à partir de l'adresse 'buffer'.

RETOUR
     Le nombre d'octets réellement lus, ou:
     0 si code de fin de fichier,
     Un numéro d'erreur négatif.

$40 Fwrite   Écriture sur un fichier

```
     LONG Fwrite(idfich,nbre,buffer)
     WORD idfich;
     LONG nbre;
     char *buffer;
```

Écrit 'nbre' octets de la mémoire à partir de l'adresse 'buffer', dans le fichier ayant comme identificateur 'idfich'.

RETOUR   Le nombre d'octets réellement écrits en cas de succès.
         Un numéro d'erreur négatif autrement.

$41 Fdelete   Effacement d'un fichier

```
     WORD Fdelete(nomfich)
     char *nomfich;
```

Efface le fichier ayant comme nom 'nomfich'.

RETOUR   0 en cas de succès
         Un numéro d'erreur négatif autrement.

$42 Fseek   Positionne le pointeur de fichier

```
     LONG Fseek(offset,idfich,mode)
     LONG offset;
     WORD idfich;
     WORD mode;
```

Positionne le pointeur dans le fichier défini par l'identificateur 'idfich'. 'offset' est un nombre signé; une valeur positive déplace le pointeur vers la fin du fichier, une valeur négative, vers le début. Le 'mode' de positionnement peut être :

```
           _____________________________________________
          | mode  |  Déplacement d'offset octets...    |
          |-------|-------------------------------------|
```

0	à partir du début du fichier	
1	à partir de la position courante	
2	à partir de la fin du fichier	

RETOUR   La position courante dans le fichier.
         Un numéro d'erreur négatif si erreur.

$43 Fattrib   Fixe ou demande les attributs d'un fichier

```
WORD Fattrib(nmfich,drap,attribs)
char *nmfich;
WORD drap;
WORD attribs;
```

'nmfich' pointe sur un chemin terminé par 0. Si 'drap' a la valeur 1, les attributs de fichier 'attribs' sont fixés (pas de valeur de retour). Si 'drap' est 0, ils sont retournés.

Les bits d'attributs sont :

masque	description
$01	fichier à lecture seule
$02	fichier transparent au répertoire
$04	fichier 'système'
$08	fichier contenant un nom de volume (11 octets)
$10	fichier sous-répertoire
$20	fichier écrit puis refermé

REMARQUE   Le bit d'archivage, $20, ne semble pas marcher comme prévu.

$45 Fdup   Duplique un identificateur de fichier

```
WORD Fdup(idfich)
WORD idfich;
```

L'identificateur 'idfich' doit être un identificateur standard (0 à 5). Fdup() retourne un identificateur non standard (supérieur ou égal à 6) qui pointe le même fichier.

RETOUR   Un identificateur ou :
EIHNDL   'idfich' n'est pas un identificateur standard
         ENHNDL   Plus d'identificateur non standard

$46 Fforce   Force un identificateur de fichier

```
Fforce(stdh,nonstdh)
WORD stdh;
WORD nonstdh;
```

Force l'identificateur standard 'stdh' à pointer le même fichier ou périphérique que l'identificateur non-standard 'nonstdh'.

RETOUR   0 si OK
         EIHNDL   identificateur invalide.

$47 Dgetpath Demande le répertoire courant

```
void Dgetpath(buf,driveno)
char *buf;
WORD driveno;
```

Le répertoire courant pour le lecteur 'driveno' est recopié dans 'buf'. Le numéro de lecteur commence à 1 pour le lecteur A:, 2 pour le B:, etc..., 0 spécifiant le disque par défaut.

REMARQUE

La taille maximum d'un chemin n'est pas limitée par le système. C'est à l'application de fournir assez de place pour le tampon. 128 octets semblent suffisants pour 8 ou 9 sous-répertoires.

--

```
$48 Malloc    Demande d'allocation mémoire
     LONG Malloc(taille)
     LONG taille;
```

Si 'taille' est -1L ($FFFFFFFF) la fonction retourne la taille du plus grand bloc libre du système. Autrement si 'taille' est différent de -1L, la fonction essaie d'allouer 'taille' octets pour le programme en cours. La fonction retourne un pointeur sur le début du bloc alloué si tout s'est bien passé, ou NULL s'il n'existait pas de bloc assez grand pour satisfaire la requête.

REMARQUE

Un programme ne peut avoir, à un instant donné plus de 20 blocs alloués par 'Malloc()'. Dépasser cette limite peut désemparer le GEMDOS. (Il est cependant possible de faire le nombre de 'Malloc()' que l'on veut à condition de les faire suivre par l'appel de la fonction Mfree(), 20 étant le nombre maximum de fragments qu'un programme peut générer).

--

```
$49 Mfree     Libération de mémoire
     WORD Mfree(adbloc)
     LONG adbloc;
```

Libère le bloc mémoire commençant à 'adbloc'. Le bloc doit être un de ceux alloués par Malloc().

RETOUR

     0 si la libération s'est bien Effectuée.
     ERROR ou le numéro d'erreur approprié sinon.

--

```
$4A Mshrink   Rétrécit la taille d'un bloc alloué
     WORD Mshrink(0,bloc,nouvtail)
     WORD) 0;
     LONG bloc;
     LONG nouvtail;
```

Rétrécit la taille d'un bloc mémoire alloué. 'bloc' pointe sur la page de base d'un programme ou sur un bloc de mémoire alloué par Malloc(), 'nouvtail' est la nouvelle taille du bloc.
Le premier argument du bloc doit être un mot nul.

RETOUR

     0   si l'ajustement de taille à été réussi.
     EIMBA   si l'adresse du bloc mémoire était invalide.
     EGSBF   si la nouvelle taille demandée était Erronée.

REMARQUE

     un bloc ne peut être que rétréci; la nouvelle taille du bloc doit forcément être inférieure à la précédente.

N.D.T.:Le compilateur 'C' Alcyon rajoute, lors de la compilation, le premier paramètre d'appel '0'. Il ne faut donc pas le mettre dans l'appel de la fonction si l'on se sert de ce compilateur.

Appel de la fonction avec le 'C' Alcyon : Mshrink(bloc,nouvtail);

--

```
$4B Pexec     Charge/Exécute un programme
     WORD Pexec(mode,ptr1,ptr2,ptr3)
     WORD mode;
     char *ptr1;
     char *ptr2;
```

char *ptr3;

Cette fonction permet différentes utilisations selon le drapeau 'mode':

mode	ptr1	ptr2	ptr3
0: charge et exécute	le fichier à exécuter	le jeu de commandes	la chaîne d' environnement
3: charge sans lancer	le fichier à charger	le jeu de commandes	la chaîne d' environnement
4: exécute uniquement	l'adr. de la page de base	(inutilisé)	(inutilisé)
5: crée une page de base	(inutilisé)	le jeu de commandes	la chaîne d' environnement

Le nom du fichier à charger ou à exécuter, 'ptr1', et la chaîne du jeu de commandes, 'ptr2', sont des chemins, terminés par 0. La chaîne d'environnement 'ptr3', est soit NULL (0L), soit un pointeur sur une structure de chaîne de la forme:

```
"chaîne1\0"
"chaîne2\0"
etc...
"chaîne3\0"
"\0"
```

La chaîne d'environnement peut être n'importe quel numéro de chaîne finie par un 0, suivie par une chaîne nulle (un simple 0). Le programme hérite d'une copie de la chaîne d'environnement parente si 'ptr3' est 'NULL'.

Le mode 0 (charge et exécute) chargera le fichier spécifié, composera sa page de base, et l'exécutera. La valeur retournée par Pexec() sera le code de sortie du processus enfant (voir Pterm0() et Pterm()).

Le mode 3 (charge sans exécuter) chargera le fichier spécifié, composera la page de base, et retournera un pointeur sur cette page de base. Le programme n'est pas exécuté.

Le mode 4 (exécute seulement) reçoit un pointeur sur une page de base. Le programme commence son exécution au début de la zone de texte, comme spécifié dans la page de base.

Le mode 5 (création d'une page de base) allouera le plus grand bloc libre de mémoire et créera la plus grande partie de sa page de base. Quelques entrées comme la taille des zones de texte/données initialisées/données non initialisées et leur adresse de base NE SONT PAS installées. Le programme appelant en a la charge.

Un programme enfant hérite des descripteurs de fichiers standards de son parent. Il emploie en fait un appel de Fdup() et de Fforce() sur les identificateurs 0 à 5.

Puisque les ressources système sont alloués lors de la création de la page de base, le processus engendré DOIT se terminer en les libérant. Ceci est particulièrement important lors de l'emploi

d'overlays. (voir 'le livre de cuisine de Pexec' pour plus de détails sur Pexec()).

$4C Pterm Termine un programme
```
     void Pterm(retcode)
     WORD retcode;
```
Termine le programme courant, ferme tous les fichiers ouverts et libère la mémoire allouée. Retourne 'retcode' au processus parent.

$4E Fsfirst Recherche la première occurence d'un fichier
```
     WORD Fsfirst(fspec,attribs)
     char *fspec;
     WORD attribs;
```
Recherche la première occurence du fichier 'fspec'. Le spécificateur de fichier peut contenir des caractères spéciaux ('?' ou '*') dans le nom de fichier mais pas dans le chemin de spécification. 'attribs' contrôle le type de fichier qu sera retourné par Fsfirst(). Son format a été décrit dans la documentation sur 'Fattrib()'.

Si 'attribs' est à zéro, les fichiers normaux seront seuls recherchés (aucun nom de volume, fichier caché, sous-répertoire ou fichier système ne sera retourné). Si 'attribs' est positionné sur les fichiers cachés ou sur les fichiers systèmes, alors ceux_ci seront pris en compte pour la recherche. Si 'attribs' est positionné pour trouver un nom de volume, alors seuls les noms de volume seront recherchés.

Lorsqu'un fichier est trouvé, une structure de 44 octets est écrite à l'emplacement pointé par le DTA.

offset	taille	contenus
0 à 20		(réservés)
21	octet	bits d'attributs
22	mot	heure de création
24	mot	date de création
26	long	taille du fichier
30	14 octets	nom et extension du fichier

Le nom de fichier et son extension se terminent par 0, et ne contiennent pas d'espaces.

RETOUR 0 si le fichier a été trouvé
EFILNF si le fichier n'a pas été trouvé, ou le numéro d'erreur approprié.

$4F Fsnext Recherche des occurences suivantes
```
     WORD Fsnext()
```
Recherche les occurences suivantes d'un fichier. La première occurence devra être recherchée par Fsfirst(). Les octets de 0 à 20 doivent rester inchangés depuis l'appel de Fsfirst() ou depuis le dernier appel de Fsnext().

RETOUR 0 si le fichier a été trouvé
ENMFIL s'il n'y a plus de fichiers trouvés, ou le numéro d'erreur approprié.

$56 Frename Renomme un fichier
```
     WORD Frename(0,ancnom,nouvnom)
     WORD) 0;
     char *ancnom;
     char *nouvnom;
```

```
Renomme  un  fichier 'ancnom' en 'nouvnom'.  Le nouveau nom ne  doit
pas déjà exister mais peut être dans un autre répertoire.

Le premier paramètre doit être 0 (mot).

RETOUR   0         si OK
         EACCDN    si le nouveau nom existe déjà
         EPTHNF    si l'ancien nom n'a pas été trouvé
         ENSAME    si l'identificateur de disque (A,B,..) n'est
                   pas le même pour les deux noms.
*------------------------------------------------------------------*
$57 Fdatime   Fixe ou demande le moment de création d'un fichier
      void Fdatime(idfich,pttemp,drap)
     WORD idfich;
     LONG pttemp;
     WORD drap;
Le  fichier  est reconnu par son identificateur  'idfich'.  'pttemp'
pointe   sur  deux mots contenant l'heure et la date en  format  DOS
(l'heure se trouve dans le premier mot,  la date dans le  second).

Si  'drap' est à 1,  la date et l'heure sont placées dans le fichier
à partir de 'pttemp'.

Si   'drap' est  à  0, la date et l'heure sont lues et placées  dans
'pttemp'.
*------------------------------------------------------------------*
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
 ******** TABLEAU DES VECTEURS D'INTERRUPTION DU 68000 ET DU 68901
*******

 Note: sur  le  ST  les  interruptions  de niveaux 1 3 et 5 ne sont
pas
 câblées (une broche du 68000 n'est pas connectée).

 Les  vecteurs  liés au MFP 68901 sont placés dans la zone des
vecteurs
 d'interruptions  utilisateurs.  Toutes les interruptions venant du
MFP
 sont prises en compte par le 68000 comme étant de niveau 6.

 L'ordre  de  priorité  interne  au  MFP  va de 15 (priorité maxi.) à
0
 (priorité  mini.). Cet  ordre  est  également celui des bits dans
tous
 registres  du  MFP  et assure donc une correspondance simple à
retenir
 entre position des bits servant à régler une interruption et niveau
de
 priorité de celle-ci.

 +-------------------------------------------------------------------
----+
 *   Niveau        Adresse       Description
*
```

```
 +---------------------------------------------------------------------
----+
 *      7              $7C            NMI (Reset)
*
 *      6              $78            MFP 68901
*
 *      5              $74
*
 *      4              $70            VBL, retour d'image
*
 *      3              $6C
*
 *      2              $68            HBL, retour de ligne
*
 *      1              $64
*
 *---------------------------------------------------------------------
----+
 * Niveau 6 : MFP 68901
*
 *
*
 *     15             $13C      Détection de moniteur monochrome
*
 *     14             $138      Indicateur de sonnerie RS232
*
 *     13             $134      Timer A (disponible)
*
 *     12             $130      Tampon de réception RS232 plein
*
 *     11             $12C      Erreur de réception RS232
*
 *     10             $128      Tampon d'émission RS232 vide
*
 *      9             $124      Erreur d'émission RS232
*
 *      8             $120      Timer B, Compteur interrup horizontales
*
 *      7             $11C      Contrôleur de disque et de DMA
*
 *      6             $118      ACIA 6850  ( MIDI et Clavier )
*
 *      5             $114      Timer C ( Horloge système à 200 Hz )
*
 *      4             $110      Timer D ( Horloge de cadencement
RS232 )   *
 *      3             $10C      Contrôleur vidéo ( fin d'opération )
*
 *      2             $108      CTS de l'E/S série
*
 *      1             $104      DCD de l'E/S série
*
 *      0             $100      Ligne BUSY de l'interface parallèle
*
 *
*
 +---------------------------------------------------------------------
----+
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
-------------------------------------------------------------------------
----
Mnemonic      :S :Size:Source      :Destination :Flags:Instr.format
:Il
..........   :  :BWL :0123456789AB:0123456789AB:XNZVC:BBBB............:
-------------+--+----+------------+------------+-----+----------------
+---
ABCD    S,D   :v :B    :0    4        :0    4        :*U*U*:
1100ddd10000msss:2
ADD     S,Dn  :^ :BWL :0123456789AB:  2345678    :*****:1101rrroooeeeeee:
2
ADDA    e,An  :  : WL :0123456789AB: 1           :-----:1101rrroooeeeeee:
2
ADDI    #,e   :  :BWL :            B:0 2345678   :*****:00000110lleeeeee:
2/6
ADDQ    #,e   :  :BWL :            B:012345678   :*****:0101###0lleeeeee:
2
ADDX    S,Dn  :v :BWL :0    4        :0    4        :*****:
1101ddd1ll00msss:2
AND     e,Dn  :^ :BWL :0 23456789AB:0 2345678   :-**00:1100rrroooeeeeee:
2
ANDI    #,e   :  :BWL :            B:0 2345678   :-**00:00000010lleeeeee:
2/6
ANDI    #,CCR :  :B    :            B:            :*****:
0000001000111100:4
ANDI    #,SR  :p : W   :            B:            :*****:
0000001001111100:4
ASw     S,Dn  :  :BWL :0            B:0           :*****:1110cccwlli00rrr:
2
ASw     e     :  : W   :             :  2345678   :*****:1110000w11eeeeee:
2
Bcc     label :  :BW   :             :            :-----:
0110CCCCDDDDDDDD:2/4
BCHG    S,e   :  :B L :0            B:0 2345678   :--*--:0000rrri01eeeeee:
2/6
BCLR    S,e   :  :B L :0            B:0 2345678   :--*--:
0000rrri10eeeeee:2/4
BRA     label :  :BW   :             :            :-----:
01100000DDDDDDDD:2/4
BSET    S,e   :  :B L :0            B:0 2345678   :--*--:
0000rrri11eeeeee:2/4
BSR     label :  :BW   :             :            :-----:
01100001DDDDDDDD:2/4
BTST    S,e   :  :B L :0            B:0 23456789AB:--*--:
0000rrri00eeeeee:2/4
CHK     e,Dn  :  : W   :0 23456789AB:0            :-*UUU:0100rrr110eeeeee:
2
CLR     e     :  :BWL :             :0 2345678   :-0100:01000010lleeeeee:
2
CMP     e,Dn  :  :BWL :0123456789AB:0            :-****:1011rrroooeeeeee:
2
CMPA    e,An  :  : WL :0123456789AB: 1           :-****:1011rrroooeeeeee:
2
CMPI    #,e   :  :BWL :0 2345678   :            B:-****:00001100lleeeeee:
2/6
CMPM    S,D   :  :BWL :   3         :   3         :-****:
1011ddd1ll001sss:2
DBcc    Dn,lbl:  : W   :0            :        8   :-----:0101CCCC11001rrr:
4
DIVS    e,Dn  :  : W   :0 23456789AB:0            :-***0:1000rrr111eeeeee:
```

```
2
DIVU    e,Dn   :  : W   :0 23456789AB:0           :-***0:1000rrr011eeeeee:
2
EOR     Dn,e   :  :BWL :0 2345678   :0 2345678   :-**00:1011rrr100eeeeee:
2
EORI    #,e    :  :BWL :           B:0 2345678   :-**00:00001010lleeeeee:
2/6
EORI    #,CCR :  :B    :           B:             :*****:
0000101000111100:4
EORI    #,SR  :p : W   :           B:             :*****:
0000101001111100:4
EXG     R,R    :  :  L :01          :01           :-----:1100rrr1mmmmmrrr:
2
EXT     Dn     :  : WL :            :0            :-**00:0100100mmm000rrr:
2
JMP     e      :  :    :            :  2  56789A :-----:0100111011eeeeee:
2
JSR     e      :  :    :            :  2  56789A :-----:0100111010eeeeee:
2
LEA     e,An   :  :  L :  2  56789A : 1           :-----:0100rrr111eeeeee:
2
LINK    An,#   :  :    : 1          :            B:-----:0100111001010rrr:
4
LSw     S,Dn   :  :BWL :0          B:0            :***0*:1110cccwlli01rrr:
2
LSw     e      :  : W  :            :  2345678   :*****:1110001w11eeeeee:
2
MOVE    e,e    :  :BWL :0123456789AB:0 2345678   :-**00:00lldeadeaseasea:
2
MOVE    CCR,e :  : W   :            :0 2345678   :-----:0100001011eeeeee:
2
MOVE    e,CCR :  : W   :0 23456789AB:             :*****:0100010011eeeeee:
2
MOVE    e,SR  :p : W   :0 23456789AB:             :*****:0100011011eeeeee:
2
MOVE    SR,e   :  : W   :            :0 2345678   :-----:0100000011eeeeee:
2
MOVE    USP,An:^p:  L : 1           : 1           :-----:010011100110wrrr:
2
MOVEA   e,An   :  : WL :0123456789AB: 1           :-----:00llrrr001eeeeee:
2
MOVEC   Rc,R   :^p:  L :            :01           :-----:010011100111101w:
               :  :    :            :             :     :Rrrrcocococococo
:4
MOVEM   RL,e   :  : WL :01          :  2 45678   :-----:010010001leeeeee:
4
MOVEM   e,RL   :  : WL :  23 56789A :01           :-----:010011001leeeeee:
4
MOVEP   Dn,D   :^ : WL :0           :     5       :-----:
0000sssmmm001ddd:4
MOVEQ   #,Dn   :  :  L :           B:0            :-
**00:0111ddd0########:2
MOVES   S,e    :^p:BWL :01          :  2345678   :-----:00001110lleeeeee:
               :  :    :            :             :     :Rrrrw00000000000
:4
MULS    e,Dn   :  : W   :0 23456789AB:0           :-**00:1100rrr111eeeeee:
2
MULU    e,Dn   :  : W   :0 23456789AB:0           :-**00:1100rrr011eeeeee:
2
NBCD    e      :  :B    :            :0 2345678   :*U*U*:0100100000eeeeee:
2
NEG     e      :  :BWL :            :0 2345678   :*****:01000100lleeeeee:
2
```

```
NEGX    e      :  :BWL :            :0 2345678   :*****:01000000lleeeeee:
2
NOP            :  :    :            :            :-----:
0100111001110001:2
NOT     e      :  :BWL :            :0 2345678   :-**00:01000110lleeeeee:
2
OR      e,Dn   :^ :BWL :0 23456789AB:0 2345678   :-**00:1000rrroooeeeeee:
2
ORI     #,e    :  :BWL :           B:0 2345678   :-
**00:00000000lleeeeee:2/6
ORI     #,CCR :  :B   :           B:            :*****:
0000000000111100:4
ORI     #,SR   :p : W  :           B:            :*****:
0000000001111100:4
PEA     e      :  :  L :            :  2  56789A :-----:0100100001eeeeee:
2
RESET          :p :    :            :            :-----:
0100111001110000:2
ROw     S,D    :  :BWL :0          B:0           :-**0*:1110cccwlli11rrr:
2
ROw     e      :  : W  :            :  2345678   :-**0*:1110011w11eeeeee:
2
ROwX    S,D    :  :BWL :0          B:0           :***0*:1110cccwlli10rrr:
2
ROwX    e      :  : W  :            :  2345678   :***0*:1110010w11eeeeee:
2
RTD     #      :  : W  :            :           B:-----:
0100111001110100:4
RTE            :p :    :            :            :*****:
0100111001110011:2
RTR            :  :    :            :            :*****:
0100111001110111:2
RTS            :  :    :            :            :-----:0100111001110101:2
SBCD    S,D    :v :B   :0   4       :0   4       :*U*U*:
1000rrr10000mrrr:2
Scc     e      :  :B   :            :0 2345678   :-----:0101CCCC11eeeeee:
2
STOP    #      :p : W  :           B:            :*****:
0100111001110010:4
SUB     e,Dn   :^ :BWL :0123456789AB:  2345678   :*****:1001rrroooeeeeee:
2
SUBA    e,An   :  : WL :0123456789AB: 1          :-----:1001rrroooeeeeee:
2
SUBI    #,e    :  :BWL :           B:0 2345678   :*****:00000100lleeeeee:
2/6
SUBQ    #,e    :  :BWL :           B:012345678   :*****:0101###1lleeeeee:
2
SUBX    S,D    :v :BWL :0   4       :0   4       :*****:
1001ddd1ll00msss:2
SWAP    Dn     :  : W  :            :0           :-**00:0100100001000rrr:
2
TAS     e      :  :B   :            :0 2345678   :-**00:0100101011eeeeee:
2
TRAP    #      :  :    :           B:            :-----:010011100100####:
2
TRAPV          :  :    :            :            :-----:
0100111001110110:2
TST     e      :  :BWL :            :0 2345678   :-**00:01001010lleeeeee:
2
UNLK    An     :  :    :            : 1          :-----:0100111001011rrr:
2
-------------+--+----+------------+------------+-----+----------------
+---
```

```
Quand on travaille avec An seules les opérations en word
ou en long word sont possibles.

m: Mode, 0=data, 1=mémoire ou 01000=Dn,01001=An,10001=les 2
s: Source            d: Destination         p: privilégiée
o: Op-mode,  B   W   L                      v: Equal addressing modes
            000 001 010 Dn+e=>Dn            ^: Operands may be swapped
            100 101 110 e+Dn=>e            l: Taille (00=B, 01=W, 10=L)
e: Effective address                        #: Donnée immédiate
c: Count or register                        w: Direction(0=droit,
1=gauche)
i: Count what? 0=imm, 1=register            C: Condition
D: Déplacement                              R: Type de registre: 0=data
U: Undefined value
1=adresse
ll: 01=B, 11=W, 10=L
=======================================================================
====
    Addr.mode  Mode Reg.: C  Bin. s/u  flags       : C  Bin. s/u
flags
---------------------------------------------------
+----------------------
 0  Dn         000  n   : CC 0100 s    C           : LS 0011 u    c+z
 1  An         001  n   : CS 0101 s    c           : LT 1101 s    n.V
+N.v
 2  (An)       010  n   : EQ 0111 u    z           : MI 1011 s    n
 3  (An)+      011  n   : F  0001 u    0           : NE 0110 u    Z
 4  -(An)      100  n   : GE 1100 s    n.v+N.V     : PL 1010 s    N
 5  d(An)      101  n   : GT 1110 s    n.v.Z+N.V.Z : T  0000 u    1
 6  d(An,Xi)   110  n   : HI 0010 s    C.Z         : VC 1000 u    V
 7  Abs.W      111  000 : LE 1111 s    z+n.V+N.v   : VS 1001 u    v
 8  Abs.L      111  001 :
 9  d(PC)      111  010 :          WHERE X = NOT x
 A  d(PC,Xi)   111  011 :
 B  Imm        111  100 :
=======================================================================
====
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

```
 *****  AFFECTATIONS  DES  VECTEURS  D'EXCEPTION  DU  68000  MOTOROLA
****

   Dans  l'ordre: Numéro  du  vecteur, adresse  en  décimal, adresse
en
   hexadécimal, descriptif de la zone, affectation.

   0   0    000  SP   RESET INITIALISATION SSP
   -   4    004  SP   RESET INITIALISATION PC
   2   8    008  SD   ERREUR BUS
   3   12   00C  SD   ERREUR D'ADRESSE
   4   16   010  SD   INSTRUCTION ILLEGALE
   5   20   014  SD   DIVISION PAR ZERO
   6   24   018  SD   INSTRUCTION CHK
   7   28   01C  SD   INSTRUCTION TRAPV
   8   32   020  SD   VIOLATION DE PRIVILEGE
```

9	36	024	SD	TRACE
10	40	028	SD	EMULATEUR LIGNE 1010
11	44	02C	SD	EMULATEUR LIGNE 1111
12	48	030	SD	(non attribué, réservé)
13	52	034	SD	(non attribué, réservé)
14	56	038	SD	(non attribué, réservé)
15	60	03C	SD	(non init. vecteur)
16-23	64	04C	SD	(non attribué, réservé)
	95	05F		---
24	96	060	SD	INTERRUPTION PARASITE
25	100	064	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 1
26	104	068	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 2
27	108	06C	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 3
28	112	070	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 4
29	116	074	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 5
30	120	078	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 6
31	124	07C	SD	AUTO VECTEUR D'INTERRUPTION NIVEAU 7
32	128	080	SD	VECTEUR D'INSTRUCTION TRAP 0
33	132	084	SD	VECTEUR D'INSTRUCTION TRAP 1
34	136	088	SD	VECTEUR D'INSTRUCTION TRAP 2
35	140	08C	SD	VECTEUR D'INSTRUCTION TRAP 3
36	144	090	SD	VECTEUR D'INSTRUCTION TRAP 4
37	148	094	SD	VECTEUR D'INSTRUCTION TRAP 5
38	152	098	SD	VECTEUR D'INSTRUCTION TRAP 6
39	156	09C	SD	VECTEUR D'INSTRUCTION TRAP 7
40	160	0A0	SD	VECTEUR D'INSTRUCTION TRAP 8
41	164	0A4	SD	VECTEUR D'INSTRUCTION TRAP 9
42	168	0A8	SD	VECTEUR D'INSTRUCTION TRAP 10
43	172	0AC	SD	VECTEUR D'INSTRUCTION TRAP 11
44	176	0B0	SD	VECTEUR D'INSTRUCTION TRAP 12
45	180	0B4	SD	VECTEUR D'INSTRUCTION TRAP 13
46	184	0B8	SD	VECTEUR D'INSTRUCTION TRAP 14
47	188	0BC	SD	VECTEUR D'INSTRUCTION TRAP 15
48-63	192	0C0	SD	(non attribués, réservés)
64-255	256	100	SD	VECTEURS D'INTERRUPTIONS UTILISATEURS

SD = zone de données superviseur
SP = zone de programme superviseur

Les vecteurs de numéros 12 à 23 et de 48 à 63 sont réservés pour des extensions futures. Aucun périphérique ne doit y être affecté.

Note: concernant le MFP, circuit générant une grande partie des interruptions dans le ST, celui-ci utilise, bien évidemment, des vecteurs d'interruptions utilisateurs. En effet, lorsqu'ici on parle d'utilisateur, ce n'est pas de vous qu'il s'agit, mais des gens qui utilise le 68000 pour fabriquer leurs machines, c'est-à-dire les gens d'ATARI, de COMMODORE ou d'APPLE!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 1

********************** L'EMULATEUR VT52 **********************

Les commandes du VT52 sont appelées en affichant le code de la touche Escape (code ASCII 27 en décimal), suivi d'un ou plusieurs paramètres.

Exemple d'utilisation avec l'affichage d'une ligne par GEMDOS 9, fonction Cconws().

```
         MOVE.L    #MESSAGE,-(SP)
         MOVE.W    #9,-(SP)
         TRAP      #1
         ADDQ.L    #6,SP

MESSAGE  DC.B      27,"E",27,"p",27,"Y",42,42,"Salut",27,"q",0
```

Efface l'écran, passe l'écriture en inverse vidéo, place le curseur sur la ligne 10, colonne 10, affiche Salut et repasse en vidéo normale.

Escape A Curseur vers le haut (s'arrête sur le bord supérieur)
Escape B Curseur vers le bas (s'arrête sur le bord inférieur)
Escape C Curseur vers la droite (s'arrête sur le bord droit)
Escape D Curseur vers la gauche (s'arrête sur le bord gauche)
Escape E Efface l'écran
Escape H Place le curseur dans le coin supérieur gauche
Escape I Curseur vers le haut (scrolling sur le bord supérieur)
Escape J Vide l'écran à partir de la position du curseur
Escape K Efface la ligne à partir de la position du curseur
Escape L Insère une ligne vide à partir de la position du curseur

Escape M Efface une ligne dans l'emplacement du curseur le reste est ramené vers le haut)

Escape Y suivi de 2 nombres. Place le curseur à une certaine position. Le premier nombre indique la ligne, le second la colonne.
Attention, il faut ajouter 32 à ces nombres pour la commande.

Escape b plus un nombre de 0 à 15. Choisit ce nombre comme registre couleur d'écriture.

Escape c plus un nombre. Idem mais pour la couleur de fond.
Escape d Vide l'écran jusqu'à l'emplacement du curseur.
Escape e Active le curseur
Escape f Désactive le curseur
Escape j Sauvegarde la position du curseur
Escape k Remet le curseur à la position sauvée avec Escape j
Escape l Efface la ligne dans laquelle se trouve le curseur
Escape o Efface la ligne jusqu'à l'emplacement du curseur
Escape p Active l'écriture en inversion vidéo
Escape q Désactive l'écriture en inversion vidéo
Escape v Active le débordement de ligne automatique
Escape w Désactive le débordement de ligne automatique

Note: Faire bien attention aux commandes: certaines utilisent des lettres majuscules, d'autres des minuscules!!!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*          COURS D'ASSEMBLEUR 68000 SUR ATARI ST                 *
*                                                                *
*             par Le Féroce Lapin (from 44E)                     *
*                                                                *
*                     Seconde série                              *
*                                                                *
*                     Cours numéro 1                             *
******************************************************************
```

Voici la seconde série de cours sur l'assembleur 68000 sur Atari. Ce cours fait suite à la première série. J'espère pour vous que cette première série a été parfaitement comprise, et que RIEN n'a été laissé de coté. Partant de ce principe, nous pouvons dire que vos bases sont bien solides, et que nous allons pouvoir allez beaucoup plus vite. La première série était destinée à vous apprendre le mécanisme de fonctionnement du 68000 et du ST au niveau de son système d'exploitation, la seconde série ne sera qu'un ensemble de ficelles, de clefs d'accès à divers choses. Si vous avez PARFAITEMENT étudiés la première série, vous pourrez tirer doucement sur ces ficelles afin de faire venir à vous les informations.

Si par contre vous 'pensez' avoir compris la première série mais que vous n'êtes pas 'certain' de tout avoir compris, il est encore temps de la relire car d'ici quelques pages vous allez commencer à vous sentir perdu, ce qui serait bien dommage!!! Pour vérifier un tout petit peu vos connaissances, voici quelques questions toutes bêtes:

```
1) MOVE.L       #$12345678,A0 puis MOVE.W #$1234,A0
        Qu'obtient-on dans A0?

2) MOVE.L  #$12345678,A0 puis MOVE.B  #$12,A0
        Qu'obtient-on dans A0?

3) Pouvez-vous expliquer concrètement ce qui se passe lorsque  je
   fais MOVE.W       #$2700,SR

4) MOVE.L       #MESSAGE,-(SP)    Que réalise cette fonction?
   MOVE.W       #9,-(SP)
   TRAP #1
   ADDQ.L       #4,SP
```

Avant de vous donner les réponses, voici la liste (non définitive) de ce qui sera traité dans cette seconde série de cours. Les traps (comment les reprogrammer), la Ligne A, le GEM, les tableaux, les programmes auto-modifiables, les macros, les inclusions de fichiers, etc... A chaque fois, le travail consistera à vous indiquer comment faire et à vous fournir une liste d'articles, d'ouvrages plus ou moins précis dans ce domaine. Il m'a semblé en effet ridicule de tartiner par exemple

50 pages sur le GEM alors que cela n'est pas susceptible d' intéresser tout le monde. Par contre il m'a semblé normal de dégrossir ce sujet et de fournir toutes les pièces nécessaires (ou, du moins, les pièces dont j'ai connaissance) afin que ceux d' entre vous qui désirent réaliser des applications de haut niveau puissent le faire. Il leur faudra bosser mais en assembleur il est courant de passer beaucoup de temps simplement à chercher de la documentation. Je vous fournis donc la liste de celle-ci, à vous de voir si vous en avez besoin. A titre indicatif, la doc que j'utilise pour GEM se nomme PRO GEM, fait environ 200 pages, et est toute en Anglais!!!! Vous vous rendez donc bien compte que faire un cours complet sur GEM grossirait de manière stupide ce cours d'assembleur!!!!

C'est ce même principe qui sera utilisé pour les différents sujets abordés dans cette seconde série. Vous trouverez d'ailleurs 2 livrets, le premier comprenant les cours eux-mêmes, le second comportant de courts listings sur les différents sujets. Attention, ces listings sont pour la plupart inutilisables sans avoir lu au préalable le cours correspondant. Pour finir je vous donnerai le même conseil que pour la première série: prenez votre temps, relisez bien chaque chapitre, faites des petits programmes en utilisant ce que vous venez d'apprendre!

Résultat du concours: Si vous avez faux à un seul truc, je vous conseille vivement de reprendre le premier cours!!!

1) On obtient #$00001234 dans A0. Ceux qui ont répondu qu'on obtenait #$12341234 ont tout faux! En effet on aurait obtenu #$12341234 si l'opération avait eu lieu sur un registre de données.
Sur un registre d'adresse pris en opérande destination, il y a extension sur le poids fort. Là, y-en-a déjà 50% qui reprennent la série 1......

2) On n'obtient rien du tout parce qu'on ne peut pas assembler!!! On ne peut travailler avec un registre d'adresse comme opérande destination que sur le format word ou long word mais pas sur le format byte.

3)$2700 cela donne en binaire %0010011100000000. Si on plaque ce nombre sur le Status Register (ceux qui croyaient que SR c'était la pile et qui on donc confondu avec SP doivent impérativement recommencer la série 1 en entier, ils sont juste prêts pour ne rien comprendre à la suite!), on se rend compte que les bits mis à 1 dans ce nombre correspondent aux bits S,I2,I1 et I0.
Comme on ne peut taper dans le SR qu'en mode superviseur, on en déduit qu'une telle opération ne peut se faire que dans ce mode. Notre MOVE conserve donc le bit superviseur dans l'état 1 puis force les bits décrivant le niveau d'interruption à 1. Le niveau d'interruption minimal pris en cours est donc le niveau 7 qui est le niveau maximum. En résumé, nous venons de bloquer, d'interdire les interruptions.

4) Cette fonction réalise 2 choses. Tout d'abord l'affichage du texte situé à l'adresse MESSAGE. C'est en effet la fonction Cconws() du gemdos. Mais cette fonction réalise aussi autre chose.... une joyeuse erreur! car en passant l'adresse puis le numéro de la fonction nous avons modifié la pile de 6 bytes (un long word pour l'adresse et un word pour le numéro de fonction) or nous ne corrigeons que de 4!!!!!

D'après vos réponses, vous pouvez continuer ou alors recommencer la série 1 avec un peu moins d'empressement. Il faut TOUT comprendre, c'est ça le secret. Si vous avez commis quelques erreurs et que voulez cependant attaquer directement la seconde série, ne soyez pas étonné d'abandonner l'assembleur dans quelques mois, découragé par des montagnes de listings auxquels vous ne comprendrez rien!

Bon courage

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Seconde série                          *
*                                                                *
*                         Cours numéro 2                         *
******************************************************************
```

Nous voici donc repartis pour de nouvelles aventures! Ce second cours aura pour sujet les TRAP et plus précisément comment les programmer soit même.Nous avions vu, dans la première série, que les traps étaient un excellent moyen d'accéder au système d'exploitation, et plus généralement d'accéder à des espaces protégés (uniquement accessible en mode Superviseur). Nous avions également étudié le passage des paramètres par la pile, ce qui nous avait servi pour réaliser des subroutines avec paramètres.

Le premier exemple va consister à changer la couleur du fond de l'écran, avec une routine fabrication maison, qui sera appelée par un trap.

D'abord la routine:
Etant donné qu'un trap est toujours exécuté en Superviseur, nous n'hésitons pas à utiliser les adresses système. La palette de couleurs du ST est située à l'adresse $FF8240. Chaque couleur étant codée sur un mot, la couleur 0 est en $FF8240, la couleur 1 en $FF8242 etc...

Nous allons faire 2 routines. Une qui mettra le fond en rouge, l'autre qui mettra le fond en vert. Les voici:

```
ROUGE   MOVE.W  #$700,$FF8240
        RTE

VERT    MOVE.W  #$070,$FF8240
        RTE
```

Une étiquette permet de les repérer (ROUGE et VERT). Les couleurs étant codées en RVB (rouge/vert/bleu. On trouve aussi RGB qui est la traduction anglaise: red/green/blue) et les niveaux varient de 0 à 7. Nous remarquons que les routines ne se terminent pas par

RTS mais par RTE. Cela signifie Return from exception. Il s'agit bien en effet d'un retour d'exception et non pas du retour d'une subroutine classique.

Petit rappel: RTE se lit "return from exception". Je vous rappelle qu'il faut TOUT lire en Anglais et pas se contenter de lire l'abré viation dont la signification est souvent assez évasive.

Voici le programme 'en entier'.

```
        MOVE.L  #MESSAGE,-(SP)  toujours sympa de se présenter
        MOVE.W  #9,-(SP)
        TRAP    #1              appel GEMDOS
        ADDQ.L  #6,SP

* Fixer les vecteur d'exception
        MOVE.L  #ROUGE,-(SP)    adresse 'routine'
        MOVE.W  #35,-(SP)       numéro de vecteur du trap #3
        MOVE.W  #5,-(SP)        fonction Setexec()
        TRAP    #13             du bios
        ADDQ.L  #8,SP

        MOVE.L  #VERT,-(SP)     adresse 'routine'
        MOVE.W  #36,-(SP)       numéro de vecteur du trap #4
        MOVE.W  #5,-(SP)        fonction Setexec()
        TRAP    #13             du bios
        ADDQ.L  #8,SP

* Les  routines sont donc maintenant accessibles par le trap 3  et
par le trap 4.
        BSR     TOUCHE
        TRAP    #3
        BSR     TOUCHE
        TRAP    #4
        BSR     TOUCHE
        TRAP    #3
        BSR     TOUCHE

        MOVE.W  #0,-(SP)
        TRAP    #1
*-------------------------------------*
ROUGE   MOVE.W  #$700,$FF8240
        RTE

VERT    MOVE.W  #$070,$FF8240
        RTE
*-------------------------------------*
TOUCHE  MOVE.W  #7,-(SP)
        TRAP    #1
        ADDQ.L  #2,SP
        RTS
*-------------------------------------*
        SECTION DATA
MESSAGE DC.B    27,"E","LES TRAPS",0
```

Facile n'est ce pas ? Et bien maintenant que vous savez mettre vos propres routines en TRAP et que vous savez également passer des paramètres à une sub routine, il ne vous reste plus qu'à faire la même chose. J'estime que vous êtes assez grand pour le faire tout seul et c'est pour cette raison que nous n'allons pas le faire ici. A vous de travailler! Une seule précaution à prendre: Une subroutine n'a besoin que de l'adresse de retour et

donc n'empile que cela. Un TRAP par contre, du fait qu'il passe en Superviseur, sauve également le Status Register. Il ne faut donc pas oublier de le prendre en compte pour calculer le saut qui vous permettra de récupérer vos paramètres passés sur la pile. L'adresse de retour est bien sûr codée sur 4 bytes et le Status Register sur 2. Il y a donc empilage de 6 bytes par le TRAP qui les dépile automatiquement au retour afin de retrouver d'où il vient et afin également de remettre comme avant le Status-Register. Il ne faudra pas non plus oublier de corriger la pile au retour.

Comme d'habitude, prenez votre temps et faites de nombreux petits essais afin de parfaitement comprendre la système.
Regardez également attentivement la fonction du Bios qui nous a servi à mettre en place nos deux routines. Si au lieu de lui fournir la nouvelle adresse pour le vecteur, nous lui passons -1, cette fonction nous retourne, dans D0.L, l'adresse actuelle correspondant à ce vecteur. Rien ne nous empêche donc de demander l'adresse utilisée par le TRAP #1 (Gemdos), de transférer cette adresse dans le trap #0 (par exemple) et de mettre notre propre routine dans le TRAP #1. Cela peut aussi vous servir pour détourner le TRAP. Par exemple pour générer automatiquement des macros. Il est possible d'imaginer ainsi un programme résident en mémoire, qui est placé à la place du trap 13 (Bios). A chaque fois qu'il y a un appel au Bios, c'est donc notre routine qui est déclenchée. Etant donné que les appels se font avec empilage des paramètres, il est tout à fait possible de savoir quelle fonction du Bios on veut appeler. Il est alors possible de réagir différemment pour certaines fonctions. Cela permet par exemple de tester des appuis sur Alternate+touches de fonction et dans ce cas, d'aller écrire des phrases dans le buffer clavier, ceci afin de générer des macros!

Note: Un trap ne peut faire appel à des traps placés 'au-dessous' de lui. Ainsi, dans un trap #1, il est tout à fait possible d'appeler un trap #13 mais l'inverse n'est pas possible.

Exemple curieux et intéressant:

```
        MOVE.W  #"A",-(SP)
        MOVE.W  #2,-(SP)
        TRAP    #1
        ADDQ.L  #4,SP

        MOVE.W  #0,-(SP)
        TRAP    #1
```

Ce court programme ne doit pas poser de problème. Nous affichons A puis nous quittons. Assemblez-le, puis passez sous MONST. Appuyez sur [Control] + P. Vous choisissez alors les préférences de MONST. Parmi celles-ci, il y a "follow traps", c'est-à-dire suivre les TRAPs qui, par défaut, est sur "NO". Tapez Y pour YES. Une fois les préférences définies, faites avancer votre programme pas à pas avec control+Z. A la différence des autres fois, lorsque vous arrivez sur le TRAP vous voyez ce qui se passe. Ne vous étonnez pas, cela va être assez long car il se passe beaucoup de chose pour afficher un caractère à l'écran. Le plus étonnant va être l'appel au trap #13. Eh oui, pour afficher un caractère le GEMDOS fait appel au Bios!!!!!

Une autre expérience tout aussi intéressante:

```
        MOVE.W  #"A",-(SP)
```

```
        MOVE.W  #2,-(SP)
        MOVE.W  #3,-(SP)
        TRAP    #13
        ADDQ.L  #6,SP

        MOVE.W  #0,-(SP)
        TRAP    #1
```

Affichage de A mais cette fois avec la fonction Bconout() du Bios. Assemblez puis passez sous MONST avec un suivi des traps. Lorsque vous arrivez dans le Bios (donc après le passage sur l'instruction TRAP #13), faites avancer pas à pas le programme mais de temps en temps taper sur la lettre V. Cela vous permet de voir l'écran. Pour revenir sous MONST tapez n'importe quelle touche. Avancer encore de quelques instructions puis retaper V etc... Au bout d'un moment vous verrez apparaître la lettre A. Réfléchissez à la notion d'écran graphique et à la notion de fontes et vous comprendrez sans mal ce qui se passe. Surprenant non ?

Quelques petites choses encore: suivez les traps du Bios, Xbios GemDos et regardez ce qui se passe au début. Vous vous rendrez compte qu'il y a sauvegarde des registres sur la pile. Seulement il n'y a pas sauvegarde de TOUS les registres! Seuls D3-D7/A3-A6 sont sauvés et donc le contenu de D2 est potentiellement écrasable par un appel au système d'exploitation. La prudence est donc conseillée. En suivant également les TRAPs vous apercevrez USP. Cela signifie User Stack Pointer c'est ainsi que l'on désigne la pile utilisateur.

Voilà, normalement les traps n'ont plus de secret pour vous. Vous devez savoir leur passer des paramètres, les reprogrammer etc ... Vous devez même vous rendre compte qu'en suivant les fonctions du système d'exploitation, on doit pouvoir découvrir comment se font telle et telle choses, et ainsi pouvoir réécrire des morceaux de routines.

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Seconde série                          *
*                                                                *
*                         Cours numéro 3                         *
******************************************************************
```

Avant de continuer à étudier ces cours, je pense qu'il est grand temps pour vous d'étudier toutes les instructions du 68000. Vous avez déjà suffisamment de bases pour réaliser de petits programmes et surtout vous devez vous être habitué avec le système qui consiste à taper seulement 2 ou 3 lignes de programmes puis à visualiser leur fonction avec MONST.

Prenez donc l'ouvrage sur le 68000 que vous avez normalement acquis, et faites le tour de toutes les instructions avec cette méthode. Observez bien les résultats des opérations de décalages (ASL, ASR etc..) Il ne s'agit pas ici de les connaître par coeur mais au moins de savoir qu'elles existent pour pouvoir, le cas échéant, les retrouver facilement, et surtout d'être capable de comprendre ce qu'elles font. Le nombre d'instructions est faible mais chacune d'elle réalise une opération bien précise qu'il convient de connaître dans ses moindres détails. Il ne suffit pas de savoir que ROR réalise une rotation vers la droite, il faut également savoir que le bit éjecté d'un coté est remis de l'autre et, qu'en plus, il est recopié dans le bit C du SR. Il faut remarquer aussi que ROR #4,D0 est accepté mais pas ROR #9,D0. Dans ce cas il faut faire 2 ROR ou MOVE.W #9,D1 puis ROR D1,D0. C'est ce genre de petits trucs qui vous bloqueront plus tard si vous n'avez pas passé quelques heures à les décortiquer!

LES INCLUSIONS DE FICHIERS

L'un des plus gros problème de l'ASSEMBLEUR se situe au niveau de la taille des listings. Si en BASIC une ligne suffit pour une opération parfois très complexe, nous avons vu qu'en ASSEMBLEUR ce n'était pas le cas. Par contre nous avons vu également que l'écriture en ASSEMBLEUR consistait à taper le bon nombre d'instruction machine, alors que le compilateur du BASIC, du C ou du PASCAL se débrouille à faire une traduction 'qui marche' et qui n'est donc pas forcément la plus économique au niveau taille et/ou au niveau vitesse.

En contre partie, nos sources ASSEMBLEUR font très rapidement des pages et des pages là, où un programmeur travaillant avec un langage 'évolué' n'aurait que quelques dizaines de lignes. Il existe cependant quelques méthodes permettant non pas d'éviter ces multiples pages, mais de réduire sensiblement la taille du listing sur lequel nous travaillons.

Deux directives vont principalement nous servir. Attention, ce ne sont pas des instructions ASSEMBLEUR, mais des ordres interprétés par l'ASSEMBLEUR. Ce sont donc, dans notre cas, des instructions 'Devpack' et non pas des instructions '68000'.

La première, INCBIN, permet d'incorporer dans le programme un fichier binaire. Un fichier binaire cela peut-être une image, de la digit, des sprites etc... ou bien un morceau de programme qui a été assemblé sous forme de fichier binaire. Voici un exemple avec une image. (listing 1 série 2)

Tout d'abord nous transférons l'adresse de l'image en A6, nous sautons l'en-tête de celle-ci pour pointer sur les couleurs qui sont mises en place avec Xbios(6), nous cherchons ensuite l'adresse de l'écran avec la fonction Xbios(3) puis, après avoir sauté la palette de couleurs de notre image, nous transférons cette image sur l'écran. Un écran fait 32000 octets. Si nous transférons long mot par long mot, nous ne devons transférer que 32000 divisé par 4 c'est à dire 8000 long mots. Comme nous utilisons une boucle DBF qui compte jusqu'à 0 compris (la boucle s'arrête quand le compteur atteint -1), il faut donc initialiser le compteur à 7999. Ensuite attente d'appui sur une touche et bye bye.

Petit exercice éducatif. Une fois ce programme assemblé, suivez le sous MONST. Lorsque Xbios(3) aura donné l'adresse de l'écran, placez la fenêtre 3 sur cette adresse puis continuez à faire avancer

pas à pas le programme, pour voir la recopie se faire. De temps en temps tapez sur la touche V afin de voir l'écran au lieu de MONST, vous pourrez ainsi suivre l'affichage 'en direct'!!!

Autre petit exercice: Le ST possède la particularité d'avoir en quelque sorte 2 écrans: l'écran sur lequel on travaille (Xbios(2)) et celui que l'on voit (Xbios(3)). Dans la plupart des cas il s'agit du même mais il est tout à fait possible de les placer à des endroits différents de la mémoire et ainsi de préparer un affichage dans xbios2 tout en montrant Xbios3. Il sera ainsi possible d'afficher rapidement Xbios2 en le transférant dans Xbios3, et ainsi, de faire des animations rapides. Essayez donc de changer un peu le listing et de mettre MOVE.W #2,-(SP) à la place de 3 pour la recherche de l'écran. Débuggez le programme et constatez!

Mais ceci nous éloigne un peu du sujet qui était l'inclusion. Pour réaliser ceci nous avons juste fourni un label de repérage qui est ici IMAGE puis l'instruction INCBIN suivi du chemin pour trouver cette image. Dans l'exemple c'est une image PI3 mais rien ne vous empêche de mettre une image PI2 ou PI1! Nous aurions très bien pu mettre 2 images l'une à la suite de l'autre, sans mettre de label pour repérer la seconde. Pour pointer dessus, sachant qu'une image DEGAS fait 32066 octets, nous aurions fait:

```
          LEA       IMAGE,A6
          ADDA.L    #32066,A6
```

Je rappelle que cela se lit: load effective address IMAGE dans A6 add address long...

Petit exercice: faire un programme qui tourne en moyenne résolution, passe en basse, affiche une image basse résolution, attend un appui sur une touche puis repasse en moyenne. Sachant utiliser les fonctions Xbios et les boucles, vous devriez être capable de faire une routine de sauvegarde de la palette et une autre de restitution de celle-ci.

Il est tout à fait possible d'inclure ainsi des fichiers très divers. Il existe pourtant plusieurs problèmes. Tout d'abord la taille du programme résultant. En effet, notre image DEGAS, de part sa taille, a grossi notre programme de 32Ko! Bien sûr il y a maintenant l'avantage de pouvoir empêcher les bidouilleurs de venir y mettre leur pieds! Encore qu'une image DEGAS fait toujours la même taille, mais il est possible d'inclure une image compactée (en mettant bien sûr une routine de décompactage dans notre programme!).

Autre problème, le temps! En effet, si l'affichage lui même est plus rapide du fait qu'il n'y a pas à aller chercher l'image sur la disquette puisque cette image est déjà en mémoire avec le programme, c'est à l'assemblage que ça pédale!!! Il faut donc passer par des mécanismes de travail bien ordonnés. Mettre en place un disque virtuel (de préférence résistant au Reset) recopier les images dedans et ensuite commencer à travailler. Vous comprendrez bien vite pourquoi les possesseurs de 520 ont tout intérêt à les faire gonfler à un méga minimum et pourquoi un lecteur externe ou mieux un disque dur devient vite indispensable!!!

Après avoir vu le mécanisme d'inclusion de fichiers, nous allons nous intéresser maintenant aux inclusions de listings. Il est en effet possible de prendre un bout de listing, de le sauver sur disquette et de demander à l'assembleur de l'inclure lors de l'assemblage. Là aussi, perte de temps à l'assemblage mais gain de

temps très appréciable lors de la création de programme. Par exemple votre routine de sauvegarde de palette: faites avec soin, elle marche bien et en fait vous en avez besoin dans tous vos programmes. Il faut donc à chaque fois la retaper et surtout consommer quelques dizaines de lignes avec cette routine. Perte de temps, possibilité d'erreur de frappe et allongement bien inutile du listing. Lorsque vous commencerez à circuler dans 10 ou 20 pages de sources à la recherche d'une variable vous commencerez à comprendre de quoi je parle, en sachant en plus que 20 pages, c'est un tout petit source assembleur...

Voyons concrètement un exemple, avec la sauvegarde de palette.

```
SAUVE_PALETTE
         MOVEM.L   D0-D4/A0-A4,-(SP)
         LEA       ANC_PAL,A4            ptn sur le lieu de
sauvegarde
         MOVE.W    #15,D4                16 couleurs

.ICI     MOVE.W    #-1,-(SP)             demande de couleurs
         MOVE.W    D4,-(SP)              numéro de la couleur
         MOVE.W    #7,-(SP)              Setcolor()
         TRAP      #14                   Xbios
         ADDQ.L    #6,SP
         MOVE.W    D0,(A4)+              sauvegarde de la couleur
         DBF       D4,.ICI               et on passe à la suivante
         MOVEM.L   (SP)+,D0-D4/A0-A4
         RTS
ANC_PAL  DC.L      0,0,0,0,0,0,0,0
REMET_PALETTE
         MOVEM.L   D0-D4/A0-A4,-(SP)
         LEA       ANC_PAL,A4            ptn sur le lieu de
sauvegarde
         MOVE.W    #15,D4                16 couleurs

.ICI     MOVE.W    (A4)+,-(SP)           demande de couleurs
         MOVE.W    D4,-(SP)              numéro de la couleur
         MOVE.W    #7,-(SP)              Setcolor()
         TRAP      #14                   Xbios
         ADDQ.L    #6,SP
         DBF       D4,.ICI               et on passe à la suivante
         MOVEM.L   (SP)+,D0-D4/A0-A4
         RTS
```

Voici donc 2 routines. Tout d'abord sachant qu'un appel à Xbios ne sauve pas les registres D0-D3 et A0-A3 nous utilisons D4 et A4 pour le compteur de couleur et l'adresse de sauvegarde, ce qui explique aussi la sauvegarde sur la pile au début des deux routines. La première sauvegarde la palette et la met à l'adresse ANC_PAL. Nous constatons que cette adresse se trouve entre les 2 routines. En effet plusieurs solutions s'offrent à nous. D'abord la plus économique en taille c'est de mettre cette réservation pour la palette dans la section BSS de notre programme en faisant ANC_PAL DS.W 16 Nous avons déjà vu que la section BSS n'occupait pas de place sur la disquette. Cependant nous avons réalisé ces routines avec en tête l'idée de les inclure, afin de gagner en facilité de programmation. En plaçant cette réservation entre les routines, elle fera partie intégrante du fichier. Il n'est cependant pas possible de la mettre en DS nous somme donc contraint de la mettre en DC. Le danger serait que notre programme essaye de lire cette partie. Faire un BSR ANC_PAL par exemple serait fatal mais nous sommes assez sérieux pour ne pas le faire, donc pas de problème...

Une fois tapé ce petit listing, sauvez le par exemple sous le nom SAUV_PAL.S. Ensuite modifiez le programme du listing 1 (celui que nous venons de voir avec l'image incluse). Juste après la fin du programme par MOVE.W #0,-(SP), TRAP #1 (donc juste avant l'inclusion de l'image), mettez

```
         INCLUDE   "A:\SAUV_PAL.S"
```

Ne mettez pas d'étiquette sur le bord gauche puisque la première étiquette c'est SAUVE_PALETTE et qu'elle est dans notre routine. Ensuite au tout début du programme, mettez BSR SAUVE_PALETTE et à la fin juste avant de quitter, mettez BSR REMET_PALETTE. Au niveau taille, votre listing est donc simplement augmenté de 3 lignes:

(BSR SAUVE_PALETTE, BSR REMET_PALETTE et INCLUDE "A:\SAUV_PAL.S")

et pourtant ce sont 24 lignes qui sont ajoutées!!!

Nous sommes donc en train de nous créer une bibliothèque. C'est une très grande partie du travail du programmeur en assembleur car de nombreuses choses reviennent souvent: initialisation par sauvegarde de la palette, passage en basse résolution et passage en Superviseur, restitution en faisant l'inverse, décompactage des images etc

De même, si vous êtes en train de réaliser un gros programme, en cours de développement ce sont des pages entières qui peuvent être incluses diminuant d'autant la taille du listing et y permettant des déplacements nettement plus aisés.

Voyons tout de suite un autre bloc qui devra maintenant faire partie de notre bibliothèque. Jusqu'à présent nous avons tapé pas mal de petits programmes, sans nous soucier de la place mémoire. Il est temps d'y penser afin de commencer à prendre conscience du fait qu'il faut programmer proprement. Imaginons que notre programme soit en mémoire mais qu'en même temps il y ait d'autres programmes dans cette mémoire. Il est bien évident que chacun ne doit s'approprier que la place mémoire dont il a besoin, afin d'en laisser le plus possible pour ces voisins. Pour cela il faut savoir que lors de l'assemblage sous forme de fichier exécutable, il y a génération d'une en-tête. Grâce à cette en-tête, au lancement de notre programme il va y avoir création de ce qu'on appelle la page de base. En voici un descriptif. Si vous désirez obtenir un maximum de renseignements sur les en-têtes de programme ou les pages de base, reportez vous aux chapitres correspondants dans la Bible ou le Livre du Développeur. Excellent chapitre sur ce sujet dans la doc officielle ATARI (document sur la structure des programmes).

* Page de base

Adresse	Description
$00	Adresse de début de cette page de base
$04	Adresse de fin de la mémoire libre
$08	Adresse du début de la section TEXTE
$0C	Taille de la section TEXTE
$10	Adresse de début de la section DATA
$14	Taille de la section DATA
$18	Adresse de début de la section BSS
$1C	Taille de la section BSS

Nous allons donc piocher ces informations, et en déduire la taille

qui devrait suffire pour notre programme. Connaissant l'emplacement de la zone d'implantation du programme, nous utiliserons la fonction 74 du GEMDOS (fonction Mshrink) qui permet, en donnant la taille désirée et l'adresse de fin, de rétrécir une zone mémoire. En effet, au lancement notre programme a pris toute la place disponible, nous devons donc la rétrécir. Notre programme a également besoin d'une pile. Au lieu de prendre celle qui est déjà en place, nous allons lui substituer la notre, dont nous pourrons régler la taille à loisir.

```
* ROUTINE DE DEMARRAGE DES PROGRAMMES

         MOVE.L    A7,A5                  prélève ptn de pile pour
prendre
                                          * les paramètres
         LEA.L     PILE,A7                impose notre pile
         MOVE.L    4(A5),A5               adresse de la page de base
de
                                          * l'ancienne pile
         MOVE.L    12(A5),D0              taille de la section texte
         ADD.L     20(A5),D0              + taille section data
         ADD.L     28(A5),D0              + taille section bss
         ADD.L     #$100,D0               + longueur de la page de
base
                                          *(256 bytes)

* Appel à la fonction MShrink() du GEMDOS (Memory Shrink)

         MOVE.L    D0,-(SP)
         MOVE.L    A5,-(SP)
         MOVE.W    #0,-(SP)
         MOVE.W    #74,-(SP)              M_shrink()
         TRAP      #1
         LEA       12(A7),A7
```

Voilà, après cette opération, notre programme n'utilise plus que la place mémoire dont il a besoin. Il ne faut pas oublier de définir la pile dans la section BSS par ceci:

```
         DS.L      256
PILE     DS.L      1
```

J'en vois qui sont surpris par cette réservation!!! Dans les exemples fournis avec DEVPACK il est marqué "stack go backwards", que je traduis librement par "moi j'avance la pile recule, comment veux tu ..." Un peu de sérieux. Nous avons vu que l'utilisation de la pile se faisait en décrémentant celle ci:

(move.w   #12,-(sp) par exemple).

Il faut donc réserver de la place AVANT l'étiquette. Pour cette raison nous notons le label et, au-dessus, la taille réellement réservée pour la pile.

Quelle taille choisir? Cela dépend de vous! Si votre programme est plein de subroutines s'appelant mutuellement et sauvant tous les registres à chaque fois, il faut prévoir assez gros.

Tapez le programme suivant. Il est évident que vous devez avoir tapé au préalable la routine de démarrage de programmes qui est un peu plus haut dans ce cours. Elle est ici incluse au début. Pas de branchement à y faire. Si, une fois assemblé, vous débuggez ce programme, vous verrez au début la routine de start. Dorénavant,

cette routine sera toujours présente au début de nos programmes mais ne sera jamais intégralement recopiée, un INCLUDE est bien plus commode! Note: Il semble que DEVAPCK se mélange parfois les pinceaux lorsque les inclusions sont nombreuses et font appel à des fichiers contenus dans des dossiers. De même il existe des problèmes avec les inclusions sur disque B lorsque celui-ci est pris comme lecteur A dans lequel on met le disque B. (je n'ai par contre pas rencontré de problème avec mon lecteur externe).

```
         INCLUDE   "A:\START.S"
         MOVE.W    #$AAAA,BIDULE
         BSR       TRUCMUCHE
         MOVE.W    BIDULE,D6
         MOVE.W    #0,-(SP)
         TRAP      #1
*-----------------------------------------------------*
TRUCMUCHE MOVEM.L  D0-D7/A0-A6,-(SP)
         BSR       MACHIN
         MOVEM.L   (SP)+,D0-D7/A0-A6
         RTS

MACHIN   MOVEM.L   D0-D7/A0-A6,-(SP)
         MOVE.L    #$12345678,D0
         MOVEM.L   (SP)+,D0-D7/A0-A6
         RTS

         SECTION BSS

BIDULE   DS.W      1
         DS.B      124
PILE     DS.L      1
         END
```

Ce programme est bien sur bidon. J'espère cependant que vous l'avez scrupuleusement tapé. Lancez le... paf 4 bombes!!!!!

Observons le à la loupe afin de comprendre pourquoi... Le start est bien mis en place et lorsque nous débuggons il est effectué sans incident. On place ensuite $AAAA dans la variable BIDULE. Activons la fenêtre 3 et pointons sur BIDULE (il suffit pour cela de faire Alternate A et de taper le nom du label en majuscule donc ici BIDULE). Avançons pas à pas, nous voyons bien BIDULE recevoir AAAA. Continuons à avancer: nous sautons dans TRUCMUCHE avec au passage sauvegarde dans la pile de l'adresse de retour (à ce propos vous pouvez faire pointer la fenêtre 3 sur PILE et regarder au dessus l'empilage des données). Ensuite nous allons sauter dans MACHIN mais là, stop!!!!!!! Suivez attentivement les explications. Juste avant d'exécuter le BSR MACHIN, faites scroller la fenêtre 1 avec la touche 'flèche vers le bas'. En effet d'après la taille du listing et celle de la fenêtre vous ne devez pas voir BIDULE. Descendez donc de 7 lignes. Normalement, la première ligne de la fenêtre doit maintenant être BSR MACHIN avec la petite flèche en face indiquant que c'est cette instruction qui va être exécutée. En bas de la fenêtre vous devez voir BIDULE avec en face DC.W $AAAA puisque c'est ce que nous y avons déposé. En dessous des ORI.B#0,D0 . En effet nous sommes dans une fenêtre qui cherche à nous montrer le désassemblage de ce qu'il y a dans le tube. Or à cette endroit il y a 0 dans le tube et cela correspond à ORI.B #0,d0; ce qui explique ces instructions. Mais ces ORI.B correspondent à quoi ? où sont-ils situés? eh bien, il sont dans notre pile puisque celle-ci est constituée d'un bloc de 124 octets entre BIDULE et PILE. Alors maintenant scrutez très attentivement BIDULE et avancez le programme d'un pas. Nous voici

maintenant sur la ligne MOVEM.L de la subroutine MACHIN. Exécutons cette ligne...

Stupeur, BIDULE est écrasé, de même que le RTS de la subroutine MACHIN qui est juste au dessus!!! Et maintenant, si nous continuons, après le second MOVEM le 68000 ne va pas tomber sur le RTS puisque celui-ci vient d'être écrasé, et notre programme va planter! Pourquoi ? eh bien, parce que nous avons essayé d'empiler 128 octets (MOVEM de trucmuche=15 registres donc 15*4=60 octets, idem pour le MOVEM de machin, auquel il faut ajouter l'adresse de retour pour trucmuche et celle de machin, total 128 octets!) alors que nous n'avions prévu qu'une pile de 124 octets.

Pour que ce programme marche sans problème, il faut donc mettre au minimum une pile de 128 octets. Faites très attention à ça, car si nous avions mis une pile de 124 octets, le programme ne se serait pas planté car il n'y aurait pas eu écrasement du RTS mais il y aurait eu écrasement de BIDULE et je suis certain que vous auriez cherché bien longtemps en vous disant " mais qu'est ce qui peut bien écraser BIDULE " surtout que cet écrasement survient à un moment où, justement, on n'utilise pas BIDULE!!!

Gardez donc toujours présent à l'esprit le principe du tube mémoire, sans oublier qu'il est plein d'instructions, de contenus de variables et que rien n'empêche de les écraser!

Encore une petite remarque sur le Start. Il est tout à fait possible de tester D0 en retour du Mshrink(). Si celui-ci est négatif, c'est qu'il y a eu erreur. Si vous savez que systématiquement vous mettez le label BYE_BYE en face du GEMDOS(0) qui termine votre programme, vous pouvez rajouter à la fin du start:

```
          TST.W     D0
          BMI        BYE_BYE
```

Une dernière précision sur les inclusions. Il existe d'autres moyens de réaliser de telles choses, par exemple d'assembler les morceaux puis de les lier avec un LINKER. Cette solution est intéressante lorsque les programmes commencent à prendre des proportions gigantesques.

Sinon, il s'agit plus d'embêtements qu'autre chose!!!! Même si certains puristes préfèrent linker:

j'édite, j'assemble, je quitte, je linke, je lance, ça plante, je débugge, j'édite etc...)

je préfère, quant à moi, la méthode de l' "include". Elle permet éventuellement d'avoir accès directement au source. Par exemple, si votre routine de sauvegarde palette plante, placer le curseur de GENST sur la ligne INCLUDE "A:\SAUV_PAL.S" puis choisissez l'option Insert File dans le menu fichier. Votre routine est maintenant sous vos yeux. Une fois que ce bloc est au point, délimitez le avec F1 et F2 puis sauver le avec F3, hop le tour est joué!

Fin du cours sur les inclusions! Commencez à fabriquez votre bibliothèque et n'hésitez pas à en faire des copies de sécurité, et méfiez vous des virus! Sur cette disquette il y a IMUN.PRG Vous le mettez en dossier Auto sur votre disquette de boot, et il vérifié toutes les disquettes que vous introduisez dans le lecteur!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Seconde série                          *
*                                                                *
*                         Cours numéro 4                         *
******************************************************************
```

Après avoir vu le principe des inclusions, nous allons nous pencher sur celui des MACROS. Ce cheminement peut vous paraître étrange, dans la mesure où nous n'abordons pas tout de suite la ligne_A (par ex.) C'est tout simplement parce que ces notions vont être abondamment reprises dans les chapitres suivants, que je me suis décidé à les aborder maintenant.

Pour expliquer ce qu'est une MACRO, nous allons partir d'un exemple simple, celui de l'affichage de BONJOUR à l'écran. Nous avons déjà vu depuis bien longtemps comment le faire avec la fonction Gemdos numéro 9.

```
        MOVE.L  #MESSAGE,-(SP)
        MOVE.W  #9,-(SP)
        TRAP    #1
        ADDQ.L  #6,SP

MESSAGE DC.B    "BONJOUR",0
```

Le problème (si on peut appeler ça un problème...) c'est qu'il faut taper 4 lignes d'instructions pour réaliser l'opération. Nous allons donc créer une grosse instruction qui va les englober toutes les 4. Micro voulant dire petit et macro voulant dire gros, nous désignerons cette 'grosse' instruction par le nom de macro-instruction ou plus couramment macro.

Il va donc falloir définir en premier lieu la macro. Dans cette exemple nous l'appellerons..... PRINT (original, non ?) Il faudra lui passer comme paramètre l'adresse de notre phrase à afficher. Comment définir cette macro ? Là se pose un problème impossible à résoudre: la méthode de définition d'une macro dépend de l'assembleur. En effet, les signes et conventions qui permettent de définir une macro sont propres à votre assembleur.

Vous en voyez tout de suite l'inconvénient: des macros définies avec Profimat sont inutilisables sous DEVPACK etc...

Voyons la définitions sous DEVPACK, ceux ne possédant pas cet assembleur ou désirant définir des macros avec un autre assembleur devront se reporter au mode d'emploi. Le principe reste cependant le même.

```
PRINT     MACRO    ADR_TXT
```

```
          MOVE.L   1,-(SP)
          MOVE.W   #9,-(SP)
          TRAP     #1
          ADDQ.L   #6,SP
          ENDM
```

Voilà notre macro définie: Tout d'abord son nom est placé tout à gauche. Ensuite on place la liste des paramètres qu'elle doit recevoir, après le mot MACRO. Dans le cas présent je l'ai appelée ADR_TXT. Pour la macro, le fait que ce nom soit le premier après le mot MACRO fait qu'il est le numéro 1. Il sera donc désigné par \1. Ensuite vient le corps de mon programme qui utilise \1 comme si c'était l'adresse de ma phrase. La définition de la macro est terminée par ENDM (END MACRO).

La définition de la macro est placée au début du programme. Parce que cette définition contient MACRO et ENDM, l'assembleur sait très bien qu'il ne faut pas l'assembler. Par contre, lorsqu'il rencontrera PRINT #MESSAGE, il saura qu'il s'agit d'une macro et ira recopier cette macro à la place de PRINT.

Voici un exemple:

```
(   recopier ici la définition de la macro print   )
VOID_INP2 MACRO
          MOVE.W   #7,-(SP)
          TRAP     #1
          ADDQ.L   #2,SP
          ENDM

END_PRG   MACRO
          MOVE.W   #0,-(SP)
          TRAP     #1
          ENDM

          PRINT    #MESSAGE
          VOID_INP2
          PRINT    #MESSAGE2
          VOID_INP2
          END_PRG

          SECTION DATA
MESSAGE   DC.B     PREMIERE PHRASE",0
MESSAGE2  DC.B     13,10,"ET LA SECONDE!!!",0
```

Tapez ce programme, puis assemblez le et lancez le. Ensuite débugger le et vous verrez que les appels de macro ont été remplacés par les textes de ces macros. Pour le moment nous n'avons vu qu'une macro avec 1 passage de paramètre. Le nombre de ces paramètres 'passables' à une macro est variable et dépend de l'assembleur. Ainsi dans DEVPACK il est possible de passer jusqu'à 36 paramètres qui seront repérés par des chiffres (0-9) puis par des lettres (A-Z ou a-z). Il est également possible d'appeler une macro à partir d'une autre macro.

Ceci étant, il existe de très nombreux inconvénients à utiliser les macros. Certains programmeurs en raffolent, j'avoue que ce n'est pas mon cas. En effet l'énorme avantage de l'assembleur réside dans le petit nombre d'instructions et donc du petit nombre de termes à retenir. dès que l'on commence à faire proliférer les macros, on se trouve coincé entre 2 feux: soit donner des noms simples et se mélanger entre eux, soit donner des noms à rallonge et se tromper souvent dans leur orthographe.

Certains poussent même la plaisanterie jusqu'à réaliser des macros pour les appels du GEMDOS, BIOS ou XBIOS, en appelant les macros par le nom des fonctions ou par le numéro. Je ne trouve pas cette solution particulièrement valable car les noms des fonctions du système d'exploitation ne sont pas très 'causant'. (une macro nommée setprt ou cprnos.....)
De plus, il faut bien reconnaître que les appels au BIOS XBIOS GEMDOS se font tous de la même manière et qu'il n'est pas difficile de s'en rappeler. Nous verrons cependant que l'utilisation des macros dans le cas de la line_A ou du GEM est une excellente solution.

A ce sujet, l'assembleur MAD_MAC est réputé pour son énorme bibliothèque de macros. Malheureusement, il subsiste un doute concernant sa disponibilité. Le club STATION Informatique affirme que c'est un DOMaine PUBlique alors qu'Atari le fournit avec le pack de développement. De toutes façons le club Station le fournissait sans sa bibliothèque de macros , ce qui en réduit considérablement l'intérêt.

Pour finir avec les macros, sachez qu'il est bien évidemment possible de se définir une bibliothèque de macros puis de l'inclure avec INCLUDE au début de votre programme. C'est ce que nous ferons avec le GEM, entre autre.

Exemple d'exercice: réaliser une macro que vous nommerez par exemple PRINT_AT et que vous appellerez ainsi

```
PRINT_AT #12,#14,#MESSAGE
```

Les 2 chiffres désignant l'emplacement auquel vous voulez que le texte soit affiché.

Une remarque: il serait tout à fait possible de ne pas mettre le # mais, à ce moment-là, il faudrait le mettre dans la macro. Par exemple

```
print     macro     1
          move.w    #\1
          move.w    #2
          trap      #1
          addq.l    #4,sp
          endm
```

Pour afficher le caractère A de code ASCII 65 il faudrait alors faire PRINT 65

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
```

```
*                        Seconde série                          *
*                                                               *
*                        Cours numéro 5                         *
*****************************************************************
```

Après ce cours sur les macros, nous allons passer à l'usage des tableaux en assembleur. J'espère qu'après chaque cours vous effectuez vous-même des recherches et des exercices, et que vous ne vous précipitez pas tout de suite sur le cours suivant. Même si les cours vous semblent maintenant courts par rapport à ceux de la première série, il n'en soulèvent pas moins le voile sur de nombreux sujets. Vous avez maintenant le niveau pour les approfondir et j'espère que vous le faites!

LES TABLEAUX
L'usage des tableaux n'est pas très répandu en assembleur au niveau des programmeurs débutants. En effet le système peut sembler assez périlleux à manipuler mais s'avère par contre très puissant et très commode!

Tout comme le principe de la pile, celui des tableaux est simple mais demande beaucoup de logique pour pouvoir fonctionner convenablement. Prenons tout d'abord un exemple simple: nous allons appuyer sur les touches de fonctions F1 jusqu'à F10 et à chaque fois nous allons afficher une lettre. Commençons par un exemple sans tableau.

Voir le listing numéro 2.

Au départ INCLUDE de la routine de démarrage de programme étudiée au début de la seconde série. Présentation avec un petit message, puis attente d'un appui sur une touche. Comme nous voulons tester les touches de fonctions et que celles-ci n'ont pas de code ASCII, nous profitons du fait que la fonction Gemdos 7 retourne dans le poids faible de D0 le code ASCII mais aussi dans le poids fort le scan code. Il n'y a d'ailleurs pas que ça de retourné, car si vous consultez la bible ST ou simplement la description des fonctions dans les dernières pages de la doc du GFA 3.00, vous y apprenez que Gemdos 7 renvoi dans les bits 0-7 le code ASCI, 16-23 le code clavier et 24-31 l'état des touches de commutations du clavier.

Petit rappel sur les codes ASCII et les scan code. Les codes ASCII (American Standard Code for Information Interchange) sont des codes sur 7 bits, qui suivent l'ordre alphabétique. Ce standard est aussi appelle Code télégraphique international numéro 5. Le 8 ème bit est un bit de parité c'est à dire qu'il ne sert qu'à la vérification de la bonne transmission du code. De part le fait que ce système est Américain, le codage ASCII ne prend pas en compte les accents ni les caractères du type c cédille. Cependant notre ordinateur n'ayant pas besoin de ce huitième bit pour vérifier les transmissions, celui-ci est utilisé pour augmenter le nombre de combinaison possible. Retour au cours 2 de la série 1, sur les chiffres magiques: avec 7 bits nous pouvons compter de 0 à 127, un bit de plus nous permet de compter de 0 à 255. Consulter un tableau des codes ASCII (bible, livre du développeur, doc GFA etc...) et vous vous rendrez compte que tous les signes 'bizarres' sont codés sur 8bits et ont donc des codes supérieurs à 127.

Cependant les codes ASCII ne suffisent pas. Imaginez que je tape sur la touche A de mon clavier. En code ASCII je vais recevoir 65. Mais si un Anglais tape sur la même touche, pour lui ce sera

un Q et il recevra le code ASCII 81. Le code ASCII ne dépend donc pas de la touche, mécaniquement parlant, mais plutôt de la lettre de cette touche. Il existe un autre codage qui lui correspond aux touches d'un point de vue mécanique. C'est ce que l'on appelle le scan-code.

Pour le scan-code, une touche possède un numéro, totalement indépendant de la lettre qui sera affectée à cette touche. Ainsi la touche A de votre clavier renvoi le scan-code $10, et il en est de même pour tous les ST du monde, que cette touche soit marquée "A", ou "Q". C'est l'emplacement sur le clavier qui compte.

Nous testons donc le scan-code de la touche sur laquelle nous avons appuyé. Si c'est Escape, nous sortons du programme. Autrement, nous allons tester s'il s'agit d'une touche de fonction. La touche F1 a pour scan-code $3B, F2->$3C, F3->$3D etc... jusqu'à F10->$44. Les scan-codes se suivant, nous testons donc notre résultat par rapport au scan-code de F1, si c'est inférieur ce n'est donc pas valable, de même par rapport au scan-code de F10: si c'est supérieur ce n'est pas valable non plus.

Ces tests étant faits, nous sommes donc en présence d'un scan-code situé entre $3B et $44, ces 2 nombres compris. $3B, cela fait 59 en décimal. Pour avoir 65 (code ASCII de A) il suffit d'ajouter 6. C'est que nous faisons ensuite. Nous obtenons donc le code ASCII de A, B, C etc... suivant la touche de fonction qui a été enfoncée. Il ne reste plus qu'à afficher cette lettre!

Imaginons maintenant que nous voulions afficher "touche F1" lorsque nous appuyons sur cette touche, "touche F2", etc.... Plusieurs solutions s'offrent à nous. En voici une première qui me vient à l'esprit et que je vais juste vous dévoiler car elle n'a pas de rapport avec les tableaux. Pour ne pas compliquer, nous n'afficherons pas "touche F10", en fait nous ne prendrons en compte que les touches F1-F9. Souvenez vous d'un des listing de la série 1. Celui qui affichait une phrase en faisant apparaître les lettres comme sur les affichages des gares ou des aéroports.Reprenez donc un peu ce listing (c'était le numéro 3) et souvenez-vous de ce qui se passait pour la phrase située à l'adresse TXT. Nous affichions cette phrase mais auparavant elle était modifiée à l'adresse colonne et à l'adresse lettre.

Dans la cas présent il suffit de faire à peu près la même chose. Préparons une phrase ainsi:

```
TXT     DC.B    "TOUCHE F"
NUMERO  DC.B    "  ",13,10,0
```

A chaque appui sur une touche de fonction, nous retirons 10 (en décimal) au scan-code, et nous mettons le résultat à l'adresse NUMERO. Ainsi le scan-code de la touche F1 ($3B donc 59 en décimal) deviendra 49 qui est le code ASCII de la lettre '1'. Nous verrons donc s'afficher 'TOUCHE F1'.

Réalisez ce programme avant de passer à la suite, cela vous fera un excellent exercice!!!!

Passons maintenant au tableau, en modifiant un peu l'affichage.

Un appui sur: affichera:

```
F1                       A
F2                       Z
F2                       E
F4                       R
F5                       T
F6                       Y
F7                       U
F8                       I
F9                       O
F10                      P
```

Première constatation: si les scan-code des touches se suivent toujours, on peut dire que le lien logique entre les lettres affichées est un peu léger...

Prenez le listing numéro 3, et commençons à l'étudier. Tout le début, jusqu'au commentaire 'la touche est valable', est strictement identique au listing précédent. Ensuite nous attaquons la partie utilisant le tableau. L'adresse de celui-ci est passée en A0 (Load Effective Adress), ensuite on retire $3B à D0 afin que celui-ci ait une valeur de 0 à 9. Le tableau que nous utilisons est composé de Words. Or, un word c'est 2 bytes, et la mémoire est composé de bytes. Pour se déplacer dans un tableau en word alors que notre unité c'est le byte, il faut donc se déplacer de 2 en 2. Il ne faut donc pas que notre 'compteur', qui est ici D0, prenne une valeur du type 0,1,2,3,4 etc... mais plutôt une valeur telle que 0,2,4,6,8...

Puisqu'à la suite de nos opérations nous avons D0 avec une valeur du type 0,1,2,3,4... il faut maintenant le multiplier par 2. Cela se fait par l'opération MULU qui se lit Multiply Unsigned. En effet c'est une multiplication non-signée, c'est-à-dire qu'elle ne prend pas en compte le signe de ce qui est multiplié, contrairement à l'opération MULS (Multiply signed).

Maintenant, observons bien cette instruction:

```
MOVE.W    0(A0,D0.W),D0
```

Il s'agit d'un MOVE donc d'un déplacement. Il se fait sur un word puisque nous avons .W Ce MOVE va prendre le D0 ième word de A0 pour le mettre dans D1. Ainsi, si nous appuyons sur F3, nous obtenons $3D. Nous retirons $3B et nous obtenons 2, nous multiplions par 2, et donc D0 vaut maintenant 4. Nous allons donc pointer sur le 4ème byte de A0 et prélever un word à partir de cet endroit. Le déplacement est en effet toujours compté avec un nombre de bytes, que le tableau soit en byte, en word ou en long. C'est un peu comme si vous vous déplaciez dans une rue avec des petites maisons, des moyennes ou des grandes, le déplacement se comptera toujours en mètres.

Mais que signifie le 0 devant la parenthèse ? Et bien c'est la valeur d'un déplacement fixe à ajouter. Prenons un exemple: Nous avons un tableau dans lequel on 'tape' suivant un chiffre qui nous est fourni par un appui sur une touche. Seulement on doit prélever des choses différentes si la touche est appuyée alors que SHIFT est enfoncée. Il est alors possible de se dire: si shift n'est pas enfoncé alors se sont les premiers éléments du tableau qui seront pris en compte, mais avec shift ce seront les éléments de la fin. On pourra alors faire:

MOVE.W   0(A0,D0.W),D1 ou bien si shift est appuyé,
MOVE.W   18(A0,D0.W),D1. Ceci revient à prendre le D0 ième word de A0, en commençant à compter à 18 bytes du début de A0.

Il faut cependant se méfier de plusieurs choses concernant les tableaux. Tout d'abord bien faire attention au type de données du tableau afin de bien modifier le 'compteur' en conséquence. Ensuite bien faire attention au fait que le premier élément c'est l'élément 0 et non pas le 1. Nous avions déjà vu dans les tous premiers cours de la série 1 les problèmes pouvant survenir lorsque l'on compte, en oubliant parfois le 0. Ce problème est d'autant plus gênant avec les tableaux que, si au lieu de retirer $3B dans mon exemple pour avoir un chiffre de 0 à 9, je n'avais retiré que $3A et ainsi obtenu 1 à 10, mon programme aurait parfaitement fonctionné. Il aurait simplement affiché n'importe quoi à la suite d'un appui sur F10. Or si vous avez un tableau de 200 éléments que vous appelez avec les touches, les touches+ shifts, +control etc... la vérification touche par touche sera peut être laissée de coté... Dans notre exemple, nous avons utilisé des words dans notre tableau. Il aurait tout à fait été possible d'utiliser des bytes.

Modifiez un peu le programme: supprimer la ligne avec MULU, et modifiez les datas. Au lieu de mettre DC.W à l'adresse TABLEAU, mettez DC.B. Pour terminer, puisque notre tableau est maintenant en bytes et non plus en words,il faut modifier l'adressage permettant de piocher dedans. Au lieu de MOVE.W 0(A0,D0.W),D1 il faut mettre maintenant MOVE.B 0(A0,D0.W),D1

Attention cependant car nous avions parlé de l'impossibilité de se servir des adresses impaires. Pourtant, dans ce dernier cas, comme notre tableau est en bytes, si D0 vaut 3, nous nous retrouvons avec une adresse impaire, et pourtant ça marche! En effet cela marche parce que nous prélevons un byte. En fait, le 68000 peut très bien prélever un byte à une adresse impaire, en revanche, ce qu'il ne peut pas faire c'est prélever une donnée plus grande (word ou long) qui commence sur une adresse impaire et qui donc chevauche les emplacements 'normaux'. Modifions encore une fois le programme. Remettez le tableau en word, et remettez l'adressage en word (MOVE.W 0(A0,D0.W),D1). L'erreur va donc venir du fait que nous avons oublié le MULU, et donc que notre compteur va être parfois impaire alors que notre tableau et notre mode d'adressage demande un compteur paire.

Assemblez et lancez. Appuyez sur F1: tout ce passe bien! Normal, D0 après retrait de $3B, vaut 0 qui est donc pair. Appuyez sur F3: même chose car D0 vaut 2. Par contre, un appui sur F2 se solde par 3 bombes et le retour à DEVPACK. Débuggons donc notre programme: alternate+D, et descendons jusqu'à la ligne:
MOVE.W 0(A0,D0.W),D1

Plaçons cette ligne en haut de la fenêtre 1 et tapons control+B Un breakpoint s'y met. Lançons avec control+R, et tapons sur la touche F2. Breakpoint, nous revoici sous MONST. En regardant la valeur de A0 nous connaissons l'adresse de notre tableau, adresse paire en l'occurrence. Par contre, si vous avez appuyez sur F2, vous devez avoir 1 comme valeur de D0, donc une valeur impaire. Avancez d'un pas sur MOVE.W 0(A0,D0.W),D1 à l'aide Control+Z. Erreur d'adresse! Il ne vous reste plus qu'à quitter par Control+C.

Bon, nous avons vu comment prélever un word ou un byte dans un tableau. Avec un tout petit peu d'intelligence vous devez être capable de prélever un long mot (au lieu de faire un MULU par 2 on en fait un par 4). Prenons un peu de recul et rappelons nous des cours précédents: nous y avons étudié le principe de ce

'tube', de cette mémoire que nous commençons à utiliser abondamment. Si vous avez un peu de mémoire justement, vous devez vous rappeler d'une remarque faite au tout début, disant qu'il fallait bien se méfier de confondre contenu du tube et adresse de ce contenu. Il est en effet tout à fait possible d'avoir

```
IMAGE          incbin         A:\MAISON.PI1"
PTN_IMAGE      DC.L           IMAGE
```

A l'adresse IMAGE, nous trouvons dans le tube l'image elle-même, mais à l'adresse PTN_IMAGE, nous trouvons un long mot, qui se trouve être l'adresse de l'image. Avec un peu d'imagination, nous pouvons donc imaginer un tableau composé de long mot, ces longs mots étant des adresses d'images, de textes, mais aussi (pourquoi pas!) de routines!!!!!!

Voici le squelette d'un programme réalisant une telle chose: Au départ, même chose que précédemment, attente d'un appui touche, vérification de la validité de la touche, on trafique pour avoir un code du type 0,1,2,3,4... puis on le mulu par 4 puisque notre tableau va être composé de long mot.

```
         LEA       TABLEAU,A0
         MOVE.L    0(A0,D0.W),A0
         JSR       (A0)
         BRA       DEBUT                  et on recommence
```

Nous faisons un JSR (jump subroutine) au lieu d'un BSR. Pourquoi? essayez, et regardez l'annexe sur les instructions pour voir les différences entre les 2 instructions!!!

Mais de quoi est composé notre tableau? eh bien, par exemple

```
TABLEAU DC.L TOUT_VERT
 DC.L    TOUT_BLEU
 DC.L    QUITTER
 DC.L    DRING
 DC.L    COUCOU
```

etc....

Toutes ces entrées étant les adresses des routines. Par exemple

```
COUCOU move.l #message,-(sp)
 move.w #9,-(sp)
 trap #1
 addq.l #6,sp
 rts
```

La routine TOUT_VERT met toute la palette en vert etc....

Il est de même possible de mettre en tableau des adresses de phrases et de passer l'adresse "piochée" à une routine qui affiche avec gemdos(9) par exemple.

Une dernière chose, qui est plus proche du système de liste que de celui de tableau, mais qui est aussi bien utile. Nous avons étudié ici des possibilités émanant toujours d'une même évidence: les données qui nous servent à pointer dans le tableau, se suivent ! Malheureusement dans de nombreux cas, celles-ci ne se suivent pas...

Voici donc une autre méthode: Imaginons le cas d'un éditeur de

texte, avec plusieurs actions possibles (effacer le texte, sauver le texte, l'imprimer, charger, écraser, scroller etc...) appelées par des combinaisons de touches. Pour être à la norme Wordstart (c'est la norme clavier utilisée par Devack: ALT+W=imprimer par exemple), j'ai d'abord relevé avec un tout petit programme, les codes renvoyés par les combinaisons de touches Ensuite j'ai réalisé une liste de ces codes, liste en word car les codes ASCII ne suffisent plus dans le cas de combinaisons de touches (il est possible bien sûr de construire la combinaison touche enfoncée/-touche de control).

```
TAB_CODE dc.w $1519,$1615,$1312,$2e03,$FFFF
```

Ensuite j'ai réalisé une liste avec les adresses de mes routines. Comme au départ je n'en avais aucune de faite, j'en ai réalisé une 'bidon', nommée JRTS et qui ne fait.... qu'RTS!

```
TAB_ROUTINE dc.l JRTS,JRTS,JRTS,JRTS
```

Ensuite j'ai fait une boucle pour lire TAB_CODE, en comparant, à chaque fois, la valeur trouvée dans le tableau avec celle de la touche. En même temps je parcours TAB_ROUTINE afin qu'au moment où je lis le 3ème élément de TAB_CODE, je sois en face du 3 ème élément de TAB_ROUTINE.

Voici le module. D7 contient le word correspondant à la touche ou à la combinaison de touche.

```
 LEA TAB_CODE,A0
 LEA TAB_ROUTINE,A1
.ICI MOVE.W (A0)+,D0
 CMP.W #$FFFF,D0
 BEQ DEBUT
 MOVE.L (A1)+,A2
 CMP.W D0,D7
 BNE .ICI
 JSR (A2)
 BRA DEBUT
```

L'adresse de la liste des codes est mise en A0 et celle des adresses de routines en A1. On prélève un word de code. C'est $FFFF? si c'est le cas, c'est donc que nous sommes en fin de liste donc on se sauve puisque la touche choisit n'est pas valable. Sinon on prélève l'adresse dans le tableau des adresses de routines. Le code du tableau est-il le même que celui de la touche ? Non, on boucle (notez ici le point devant le label. Cela indique que le label est local. L'assembleur le rapportera au label le plus proche portant ce nom. Il est ainsi possible d'avoir plusieurs label .ICI dans un programme, ou tout autre nom pourvu qu'il soit précédé d'un point. Dans le cas de petites boucles par exemple, cela évite de chercher des noms de labels tordus!!!).

Puisque le code est identique à celui de la touche, on saute vers la routine, et au retour, on recommence.

Les avantages de cette méthode sont multiples. Tout d'abord la petite taille de la routine de test: s'il avait fallu réaliser des tests du genre:

```
 cmp.w #$1519,d7
 bne.s .ici1
 bsr truc
```

```
 bra début
.ici1 cmp.w #$1615,d7
 bne.s .ici2
 bsr truc2
 bra début
.ici2
```
etc.... la taille aurait été supérieure, surtout que dans l'exemple il n'y a que 4 codes... Imaginez avec une trentaine!!! L'autre avantage concerne la mise au point du programme. En effet rien n'empêche de prévoir plein de routines mais de simplement mettre l'adresse d'un RTS, et progressivement de construire ces routines. Rien n'empêchera pour autant le programme de marcher. De même, le système du flag de fin ($FFFF) permet de rajouter très facilement des codes et donc des routines.

Voila pour les listes et tableaux!! les applications sont innombrables et permettent souvent en peu de lignes des tests, des recherches, des calculs, des conversions ou des branchements réalisables autrement mais au prix de difficultés immenses!

Voici des idées d'applications:

codage de texte (utilisation du code ASCII pour pointer dans un tableau donnant la valeur à substituer)

animation (tableau parcouru séquentiellement donnant les adresses des images à afficher)

gestion de touches

gestion de souris (tableau avec les coordonnées des zones écran)
menu déroulant

etc...

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Seconde série                          *
*                                                                *
*                         Cours numéro 6                         *
******************************************************************
```

LES AUTO-MODIFICATIONS

Autre chose simple à utiliser et qui facilite beaucoup la programmation: les programmes auto-modifiables. Comme tous les sujets abordés jusqu'ici, celui n'est pas compliqué mais demande un peu d'attention. Je dois cependant avouer que le première fois que

j'ai rencontré une telle chose dans un listing, de nombreuses heures m'ont été nécessaires avant de comprendre ! La principale difficulté résidant non pas dans la compréhension du sujet lui-même mais plutôt dans le choix de la méthode d'explication, j'espère que celle-ci vous donnera satisfaction!

Il est tout à fait possible d'imaginer une addition avec des variables. Par exemple A=1, B=2 pour une opération du genre A+B=C Nous imaginons également sans mal que les valeurs de A et B puissent changer en cours de programme pour devenir par exemple A=2 et B=3 ce qui laisse notre opération A+B=C toujours aussi valable. Mais, comment faire pour que cette opération A+B=C devienne tout à coup A-B=C ou encore A/B=C ?

Là se fait toute la différence entre un langage évolué et l'assembleur. Nous avons vu, dans les premiers cours, que l'assembleur ne faisait que traduire en chiffres les instructions. A la différence des compilateurs qui 'arrangent' les instructions, l'assembleur ne réalise qu'une traduction, instruction par instruction. Nous nous retrouvons donc avec une suite de chiffres, ces chiffres étant dans le 'tube'. Tout comme nous avons écrit dans le tube pour modifier des valeurs données à des variables, il est donc tout a fait possible d'écrire dans le tube pour modifier les chiffres qui sont en fait des instructions. Il est évident que la prudence s'impose car les chiffres que nous allons écrire doivent être reconnus par le 68000 comme une nouvelle instruction et non pas comme n'importe quoi, ce qui conduirait à une erreur. Voyons concrètement un exemple simple.Nous avons une liste de lettres codée en word, et nous voulons afficher ces lettres les unes après les autres.

Voici un programme réalisant cette opération.

```
         INCLUDE    "B:\START.S"
         LEA        TABLEAU,A6      dans A6 car GEMDOS n'y touche pas
DEBUT    MOVE.W     (A6)+,D0        prélève le word
         CMP.W      #$FFFF,D0       c'est le flag de fin?
         BEQ        FIN             oui, bye bye
         MOVE.W     D0,-(SP)        non, donc le passe sur la pile
         MOVE.W     #2,-(SP)        pour l'afficher
         TRAP       #1
         ADDQ.L     #4,SP
         MOVE.W     #7,-(SP)        attend un appui touche
         TRAP       #1
         ADDQ.L     #2,SP
         BRA        DEBUT           et on recommence
FIN      MOVE.W     #0,-(SP)
         TRAP       #1
*---------------------------------*
         SECTION DATA
TABLEAU  DC.W       65,66,67,68,69,70,$FFFF
         SECTION BSS
         DS.L       100
PILE     DS.L       1
         END
```

Imaginons maintenant que cet affichage soit dans une subroutine, et que nous voulions afficher une lettre à chaque appel de cette subroutine: On attend un appui sur une touche, si c'est 'espace', alors on s'en va, sinon on saute à la routine qui affiche un caractère puis revient. Voici un premier essai:

```
         INCLUDE    "B:\START.S"
DEBUT    MOVE.W     #7,-(SP)
         TRAP       #1
         ADDQ.L     #2,SP
         CMP.W      #" ",D0
         BEQ        FIN
         BSR        AFFICHE
         BRA        DEBUT

FIN      MOVE.W     #0,-(SP)
         TRAP       #1
*-------------------------------*
AFFICHE  LEA        TABLEAU,A6      adresse du tableau
         MOVE.W     A6)+,D0         prélève le word
         MOVE.W     D0,-(SP)        le passe sur la pile
         MOVE.W     #2,-(SP)        pour l'afficher
         TRAP       #1
         ADDQ.L     #4,SP
         RTS                        puis on remonte
*---------------------------------*
         SECTION DATA
TABLEAU  DC.W       65,66,67,68,69,70,$FFFF
         SECTION BSS
         DS.L       100
PILE     DS.L       1
         END
```

Assemblez et lancez le programme. Constatation: à chaque appui sur une touche, on obtient un 'A' mais pas les autres lettres!!! Evidemment, puisqu'à chaque fois que nous sautons dans notre subroutine AFFICHE, celle-ci recharge l'adresse du tableau. Le caractère prélevé est donc toujours le premier. Pour éviter cela, il faut donc créer un pointeur qui avancera dans ce tableau. Dans notre exemple, il suffisait en fait de placer le LEA TABLEAU,A6 au début du programme. A6 n'étant modifié par personne, cela aurait fonctionné.... jusqu'au 7ème appui sur une touche, A6 pointant alors hors du tableau ! De plus, nous sommes ici pour apprendre et nous envisageons donc le cas où, en dehors de la routine, tous les registres sont modifiés! Impossible donc de garder A6 comme pointeur. Voici donc la routine AFFICHE modifiée:

```
AFFICHE  MOVEA.L    PTN_TAB,A0
         MOVE.W     (A0)+,D0
         CMP.W      #$FFFF,D0
         BNE        .ICI
         LEA        TABLEAU,A0
         MOVE.L     A0,PTN_TAB
         BRA        AFFICHE
.ICI     MOVE.L     A0,PTN_TAB
         MOVE.W     D0,-(SP)
         MOVE.W     #2,-(SP)
         TRAP       #1
         ADDQ.L     #4,SP
         RTS
```

De plus il faut rajouter après le INCLUDE (donc avant le label début)

```
         LEA        TABLEAU,A0
         MOVE.L     A0,PTN_TAB
```

et en section BSS

```
PTN_TAB  DS.L       1
```

Un peu d'analyse après ces changements! Tout d'abord nous constatons avec bonheur que ça marche! Au début nous mettons en place un pointeur.

```
          LEA       TABLEAU,A0     met l'adresse du tableau en A0
          MOVE.L    A0,PTN_TAB     et la sauve dans PTN_TAB
```

Nous avons donc maintenant dans le tube en face de l'étiquette PTN_TAB un long mot, ce long mot étant l'adresse du début du tableau. Ensuite dans la routine, nous prélevons cette adresse. Ici une petite remarque s'impose car la confusion est fréquente: Si nous avons:

```
IMAGE     INCBIN    "A:\MAISON.PI1"
```

et que nous voulons travailler avec cette image, nous ferons

```
          LEA       IMAGE,A0
```

A0 pointera alors sur l'image. Par contre si nous avons :

```
PTN_IMAG DC.L       IMAGE
```

C'est-à-dire une étiquette pour un long mot se trouvant être l'adresse de l'image, en faisant LEA PTN_IMAGE,A0 nous ne récupérons pas en A0 l'adresse de l'image mais en fait l'adresse de l'adresse de l'image! Pour récupérer directement un pointeur sur l'image il faut faire:

```
          MOVEA.L   PTN_IMAGE,A0
```

Cependant, pour récupérer l'adresse du tableau il aurait également été possible de faire:

```
          MOVEA.L   #TABLEAU,A0
```

Ceci dit, continuons notre exploration: Dans PTN_TAB nous avons donc l'adresse du début du tableau. Attente d'un appui touche, hop on saute dans la routine. Transfert l'adresse contenue dans PTN_TAB dans A0 puis on prélève le word contenu dans le tube à cette adresse et on le met en D0. Comme nous avons réalisé cette opération avec (A0)+, A0 point donc maintenant sur le prochain word du tableau. Testons si le word prélevé est $FFFF, ce qui indiquerait la fin du tableau. Si ce n'est pas le cas, on saute à .ICI et on sauve la nouvelle valeur de A0 dans PTN_TAB.

Si le word prélevé est $FFFF, on recharge PTN_TAB avec l'adresse du haut du tableau, et c'est reparti comme en 14!!!

Ce système de pointeur, très fréquemment utilisé, est simple d'emploi et bien commode! Voyons cependant une autre méthode, plus tordue! Supprimons tout d'abord la routine AFFICHE et remplaçons la par la suivante:

```
AFFICHE   MOVEA.L   #TABLEAU,A0
          MOVE.W    (A0)+,D0
          MOVE.W    D0,-(SP)
          MOVE.W    #2,-(SP)
          TRAP      #1
          ADDQ.L    #4,SP
          RTS
```

Réassemblons et lançons. Il est bien évident que cela ne marche plus puisqu'à chaque appel de la routine, nous rechargeons A0 avec l'adresse du TABLEAU, donc le word prélevé sera toujours le premier du tableau. Passons sous MONST avec Alt+D. Descendons sur le label AFFICHE. Nous trouvons en face MOVEA.L #TABLEAU,A0 etc.... Quittons avec control+C puis réassemblons, mais attention avant de cliquer sur 'assembler', jetons un coup d'oeil sur les options. Nous avons par défaut DEBUG INFO indiquant Extend. Cela signifie que les noms des labels vont être incorporés dans le programme. Cela nous permet de retrouver les noms de ces labels lorsque nous sommes sous MONST. Choisissons l'option NONE pour DEBUG INFO assemblons et repassons sous MONST.

Surprise, les noms des labels ont disparu et sont remplacés par des chiffres. C'est logique puisque, de toute façon, l'assembleur traduit notre source en chiffres. Cherchons notre routine AFFICHAGE. C'est un peu plus dur puisque son étiquette n'est plus visible! Pour se repérer, on peut chercher au début (après le start) CMP.W #$20,D0 qui est la comparaison avec la touche espace après l'appui touche. Ensuite, un BEQ vers la fin et le BSR vers notre routine. Relevons l'adresse située en face du BSR et allons-y. La première ligne de notre routine c'est MOVEA.L #$XXXXXXX,A0 XXXXXXX étant l'adresse du tableau. Je rappelle que sur un 68000 le programme peut se trouver n'importe où en mémoire, cette adresse sera donc différente suivant les machines. Pour ma part c'est $924C6. J'active la fenêtre 3 avec Alt+3 puis avec alt+a je demande à la fenêtre de se positionner sur cette adresse. MONST m'affiche au centre les codes ASCII des lettres de mon tableau ($41,$42 etc...) et à droite ces lettres en 'texte'.

En avançant dans cette routine d'affichage, je vais donc mettre (pour moi) $924C6 en A0, cette adresse étant celle pointant sur le 'A' du tableau. Ce qui m'intéresserait, c'est que, la prochaine fois, cela me permette de pointer sur le 'B'. Pour cela il faudrait avoir:

```
         MOVEA.L   #$924C6,A0     pour le 'A'
```

et ensuite

```
         MOVEA.L   #$924C8,A0     pour le 'B'.
```

Les lettres étant sous forme de word dans mon tableau il faut une avance de 2 !

Retournons dans la fenêtre 2, en face de ce MOVEA.L, regardons l'adresse à laquelle il se trouve (colonne de gauche), notons cette adresse, et notons également l'adresse de l'instruction suivante (MOVE.W (A0)+,D0). Activons la fenêtre 3, et plaçons nous à l'adresse du MOVEA.L.

Dans mon cas, et puisque j'avais:

```
         MOVEA.L   #$924C6,A0     je trouve 207C 0009 24C6
```

J'en déduis que ces 3 words constituent la représentation de mon instruction MOVEA.L, puisque l'adresse du word suivant correspond à celle de l'instruction suivante. Or, je retrouve dans ce codage, l'adresse de mon tableau. Avec un peu d'imagination, je conçois aisément qu'il est possible d'écrire directement dans le 'tube' et par exemple de modifier le word qui a pour valeur actuelle 24C6. Si je lui ajoute 2, mon instruction deviendra 207C 0009 24C8 ce qui reviendra à MOVEA.L #$924C8,A0 et qui me fera pointer sur le second word du tableau!!!!!!!!

Voici donc la version auto-modifiable de la routine AFFICHE.

```
AFFICHE MOVEA.L    #TABLEAU,A0
        MOVE.W     A0),D0
        CMP.W      #$FFFF,D0
        BNE        ICI
        MOVE.L     #TABLEAU,AFFICHE+2
        BRA        AFFICHE
.ICI    ADD.W      #2,AFFICHE+4
        MOVE.W     D0,-(SP)
        MOVE.W     #2,-(SP)
        TRAP       #1
        ADDQ.L     #4,SP
        RTS
```

Note: PTN_TAB ne nous sert plus, et de même le LEA tableau du début.

Assemblez avec NONE en DEBUG INFO, puis passez sous MONST, faites avancer pas à pas et regardez la ligne

```
        MOVEA.L    #TABLEAU,A0    se modifier!
```

Expliquons bien clairement ce qui se passe.

Nous plaçons TABLEAU en A0 puis nous prélevons le word. Admettons tout d'abord qu'il ne s'agisse pas de $FFFF, nous sautons donc à .ICI. Il faut donc ajouter 2 pour augmenter l'adresse et pointer la prochaine fois sur la seconde lettre du tableau. Nous avons vu qu'en étant codée la ligne MOVEA.L etc... tient sur 3 words donc 6 bytes. L'ajout de 2 doit donc porter sur le 3ème word. Le début de ce word c'est le byte 4. Pour cette raison nous donnons comme destination de l'addition AFFICHE+4.

Si nous avions prélevé $FFFF, il aurait fallu réinitialiser notre ligne MOVEA.L avec

```
        MOVE.L     #TABLEAU,AFFICHE+2.
```

Pourquoi +2 ? Parce que l'adresse de tableau est un long mot et que, dans le codage de l'instruction, cela commence sur le second word. Il faut donc sauter un seul word c'est-à-dire 2 bytes.

Dans le même ordre de chose, il est tout à fait possible de modifier plus profondément un programme. En voici un exemple flagrant. (voir listing numéro 4)

Sachant que l'instruction RTS (Return from subroutine) est codée avec $4E75 et que l'instruction NOP (No operation) est codée par $4E71, en plaçant un NOP ou un RTS, on change en fait la fin de la routine. NOP ne fait rien du tout. C'est une opération qui est bidon dans le sens où rien ne change, mais cette instruction consomme un peu de temps. Elle nous servira donc pour réaliser des petites attentes (bien utile pour des effets graphiques par exemple).

Suivez bien le déroulement de ce programme sous MONST afin de voir les modifications se faire. Un cas plus complexe:

```
         MOVE.W    #23,D0
         MOVE.W    #25,D1
VARIANTE ADD.W     D0,D1
         MULU.W    #3,D1
         SUB.W     #6,D1
```

```
         MOVE.W    D1,D5
```

Après assemblage de ce petit morceau de programme, passez sous MONST et jetez un coup d'oeil à la fenêtre 3. En pointant sur VARIANTE et en regardant les adresses en face des instructions, on en déduit que:

```
         ADD.W     D0,D1      est converti en $D240
         MULU.W    #3,D1      est converti en $C2FC $0003
         SUB.W     #6,D1      est converti en $0441 $0006
```

Si nous prenons maintenant:

```
         MOVE.W    #23,D0
         MOVE.W    #25,D1
VARIANTE MULU.W    D0,D1
         SUB.W     #8,D1
         ADD.W     #4,D0
         MOVE.W    D1,D5
```

Nous assemblons, passons sous MONST:

```
         MULU.W    D0,D1      est converti en $C2C0
         SUB.W     #8,D1      est converti en $0441 $0008
         ADD.W     #4,D0      est converti en $0640 $0004
```

Donc, si dans un programme utilisant cette 'routine' je fais

```
         LEA       VARIANTE,A0
         MOVE.W    #$D240,(A0)+
         MOVE.L    #$C2FC0003,(A0)+
         MOVE.L    #$04410006,(A0)+
```

J'obtiendrai la première version:

```
         ADD.W     D0,D1;
         MULU.W    #3,D1;
         SUB.W     #6,D1
```

alors que si je fais:

```
         LEA       VARIANTE,A0
         MOVE.W    #$C2C0,(A0)+
         MOVE.L    #$04410008,(A0)+
         MOVE.L    #$06400004,(A0)+
```

j'obtiendrai la seconde version!

Essayez avec le programme ci-après, en le suivant sous MONST:
Note: ce programme n'a pas de fin donc quitter avec Control+C:

```
         LEA       VARIANTE,A0
         MOVE.W    #$D240,(A0)+
         MOVE.L    #$C2FC0003,(A0)+
         MOVE.L    #$04410006,(A0)+

         LEA       VARIANTE,A0
         MOVE.W    #$C2C0,(A0)+
         MOVE.L    #$04410008,(A0)+
         MOVE.L    #$06400004,(A0)+

         MOVE.W    #23,D0
         MOVE.W    #25,D1
VARIANTE MULU.W    D0,D1
```

```
         SUB.W     #8,D1
         ADD.W     #4,D0
         MOVE.W    D1,D5
         END
```

Remarques: Il est tout à fait possible d'envisager plus de 2 versions d'une même partie de programme. Si les tailles de ces différentes versions diffèrent, ce n'est pas grave car il est toujours possible de combler avec des NOP. Les applications de ce genre de 'ruse' peuvent être assez nombreuses: raccourcissement de programmes, rapidité (une routine devant être appelée 15000 fois aura tout intérêt à être modifiée avant, au lieu d'y incorporer des tests, par exemple), modifications aléatoires des routines de protection (un coup, j'en mets une en place la prochaine fois, j'en mettrai une autre...)....

Faites cependant bien attention, car une erreur d'un chiffre et le nouveau code mis en place ne voudra plus rien dire du tout! Faites également attention à vos commentaires car là, ils deviennent hyper importants, étant donné que le listing que vous avez sous les yeux ne sera pas forcément celui qui sera exécuté!!!!!!!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*                COURS D'ASSEMBLEUR 68000 SUR ATARI ST           *
*                                                                *
*                  par Le Féroce Lapin (from 44E)                *
*                                                                *
*                           Seconde série                        *
*                                                                *
*                           Cours numéro 7                       *
******************************************************************
```

Inutile de vous complimenter inutilement, mais si vous avez suivi bien tranquillement les cours depuis le début, si vous avez pris soin de vous exercer avec des petites applications, vous devez déjà avoir un niveau correct en assembleur! N'oubliez donc pas les commandements principaux: bien commenter les programmes, réfléchir simplement, utiliser papier et crayon etc... Plus vous avancerez et plus vos listings seront longs, et plus une programmation propre sera indispensable!!! De même cherchez par vous même avant de poser dès questions, cela évitera bien souvent de demander n'importe quoi...

Dans ce cours ainsi que dans les 2 suivants, nous aborderons2 sujets tabous: la Ligne A et le GEM.

Concernant ces 2 sujets, les critiques vont bon train: la ligne A c'est pas commode, c'est nul et le GEM, pouah!!! Après pas mal de temps passé à discuter et à écouter les opinions de plusieurs personnes, je suis arrivé à la conclusion suivante: Tout d'abord les critiqueurs sont rarement les programmeurs Ligne A ou GEM. Ensuite, imaginons un cours instant notre ST sans GEM et préparons nous à faire un effort, GEM nous le rend bien! En dernier lieu je

dirais que ces cours sont destinés à faire de vous des programmeurs et pas des bidouilleurs. Vous devez donc avoir connaissances des possibilités de la ligne A et de celles du GEM. Ne pensez pas cependant trouver dans ces cours la totalité des commandes GEM. Ces cours sont destinés à levez un coin du voile, et si possible à vous donnez de quoi continuer vos recherches.

En tout cas je peux vous assurer que la ligne A et le GEM ne sont pas plus dur qu'autre choses (D'ailleurs y a t-il quelques chose de dur en assembleur?????). Il y a également un avantage non-négligeable concernant la ligne A et surtout GEM: Jusqu'à maintenant, nous avons utilisé des instructions assembleur 68000, et il semblait évident que dans un ouvrage sur le PASCAL, le BASIC GFA ou le C, nous aurions eu du mal à trouver des précisions sur l'instruction MOVEM.L par exemple ! Concernant la ligne A, mais surtout GEM, nous allons utiliser principalement des macros (vous voyez pourquoi ce sujet a été abordé avant!). Or ces macros ont chacune un nom, et ce nom est un nom 'GEM' et non pas relatif à tel ou tel assembleur!! Ainsi ouvrir une fenêtre pour la faire apparaître sur l'écran, se fera avec une macro nommée WIND_OPEN. L'avantage c'est que cette fonction GEM est utilisée en Pascal, en C etc... Pour cette raison, vous pouvez chercher de la documentation sur GEM quelque soit le langage utilisé dans cette documentation. Dans 99% des cas ce sera le C.

Pour ma part mes connaissances en C sont réduites et pourtant, une grande partie des documents GEM avec lesquels je travaille, fait référence à ce langage. Surtout ne vous contentez pas des ouvrages traitant du GEM en assembleur car vous serez bien vite dégoûté!!

Idem pour la ligne A, les paramètres nécessaires ne dépendent pas du langage!!

Un autre avantage très important, surtout pour le GEM, concerne les formats de données. Ayant la chance de pouvoir travailler sur des Macs, de nombreuses choses y rendent la manipulation des programmes plus aisée que sur ST. Par exemple on charge un logiciel de dessin, on fait un petit gribouillis, ensuite on charge n'importe quel traitement de texte et hop on y colle notre gribouillis. Sur ST il y a 99% de chance que le format de l'un ne soit pas reconnu par l'autre! Si on jette un coup d'oeil vers le monde PC, on se rend compte qu'il n'existe pas de format normalisé pour ce type de machine, et que c'est l'anarchie à ce niveau. De tels formats normalisés existent sur Mac, ils sont respectés, et les rares softs ne se pliant pas à cette norme sont voués à un formatage rapide de la disquette!!! Le comble c'est que sur ST de telles normes de formats existent, mais elles ne sont pas respectées....

En fait l'utilisation de la ligne A et surtout celle du GEM, doivent nous amener à une réflexion plus globale sur ce qu'est la programmation. Raymond Loewy (1893-1986), que l'on peut considérer comme le père du design industriel, a eu beaucoup de réflexions sur ce qu'il faut donner au consommateur. Dans notre langage nous parlons d'utilisateur de softs mais c'est en fait de la même personne qu'il s'agit. Comparons un soft Mac et un soft ST. L'écran est allumé, le soft est chargé, et la seule chose que nous faisons, c'est promener notre oeil sur les icônes et la souris dans la barre de menu. Difficile de juger des qualités respectives des 2 softs sans aller plus loin. Pourtant, la simple disposition des entrées dans les menus, le positionnement des icônes, le faible nombre de celles-ci mais en même temps leur impression de clarté donne dans 99% des cas l'avantage au Mac Intosh. Sur le ST,

les menus dégueulent de partout (surtout dans les softs récents), et les icônes recouvrent un bureau qui n'en demande pas tant! Ceci est à rapprocher d'une réflexion de Raymond Loewy que je vous demande de méditer: "Un véhicule aérodynamique bien dessiné donne une impression de vitesse, même quand il est arrêté". Inutile donc d'en rajouter à la pelle, les 14524874 fonctions du softs ST sont simplement des délires de programmeurs alors que les 20 fonctions du softs Mac ont été pensées pour l'utilisateur. Lorsque Raymond Loewy a été demandé pour faire le nouveau logo de la société Shell, ainsi que tout le design des stations services, il a simplement dit à ses employeurs: "Vos clients, ce ne sont pas les automobiles, ce sont les automobilistes". Evident, sauf qu'à l'époque la Shell vendait pour les automobiles, sans s'en rendre compte!

Eh bien, sur ST les programmeurs programment pour le ST (ou pour eux!) mais très rarement pour les utilisateurs...

Tout ceci étant dit, nous allons aborder maintenant la ligne A, en espérant cependant que cela vous fera réfléchir lorsque vous concevrez vos programmes!

LA LIGNE A

Doc officielle Atari, Chapitre Ligne A: Bon descriptif des différentes fonctions, assez brut mais efficace.

Bible ST: Bon descriptif des fonctions Livre du développeur: idem mais en plus descriptif des données dont on peut déduire l'emplacement avec la ligne A.

Doc du GFA 3.00: Assez étrange. Si les fonctions ligne A du GFA sont appelées de la même manière qu'en assembleur, le concepteur de ce langage a réussi à 'inventer' des fonctions en clamant bien haut qu'il s'agit de fonction Ligne A !!! Peut-être pour que le GFA soit encore moins compatible... A ce propos, il semblerait que la ligne A ne soit plus officiellement supportée par le TT.... A voir...

Dans le premier kilo de mémoire de notre machine, nous avons un bon paquet d'adresses utilisées par le microprocesseur. Nous avons passé pas mal de temps à éplucher le principe de fonctionnement de ce kilo, qui sert au 68000 de 'relais' afin qu'il trouve l'adresse de la routine qu'il doit exécuter pour telle ou telle raison. J'espère de tout coeur que vous avez parfaitement assimilé ce principe car pour la ligne A, le GEM, les interruptions etc... c'est encore et toujours ce même principe qui est appliqué ! Prenez la feuille qui vous a été fournie avec les cours de la série 1, et qui montre ce premier kilo. Le vecteur 10 (adresse 40 en décimal et $28 en hexa.) possède comme 'description': émulateur ligne 1010. Convertissons 1010 en hexadécimal, nous obtenons.... $A ! Première constatation, si ce vecteur est dans le premier kilo de mémoire, c'est donc que l'émulateur ligne A n'est pas une particularité du ST mais bien du 68000. Ces vecteurs vont donc nous donner accès à une ou des routines, en utilisant toujours le même principe:

Ordre au 68000, celui-ci analyse l'ordre, saute dans le premier kilo de mémoire à l'adresse correspondant à cet ordre, y trouve l'adresse de la routine à exécuter, il ne lui reste plus qu'à sauter à celle-ci.

Comment donc appeler les routines accessibles par l'intermédiaire

de l'émulateur ligne A, que font ces routines, et comment 'discuter' avec elles, c'est à dire comment leur passer d'éventuels paramètres ?

Dans le cas de l'Atari ST, les routines accessibles par le biais de l'émulateur ligne A sont des routines graphiques. Ces routines sont les routines graphiques de base de notre machine. Elles sont utilisées par le GEM. Leur accès est rustique et mal commode mais la commodité n'est pas le but recherché. Généralement en assembleur on recherche souvent à faire soit même ses routines (c'est toujours mieux que celles des copains) Il faut cependant noter que dans beaucoup de cas, une bonne utilisation de la Ligne A ou du Gem est amplement suffisante. De plus, c'est bien souvent une bonne connaissance de ces interfaces qui vous permettra d'améliorer vos routines. Une raison supplémentaire réside dans la facilité de mise en oeuvre et dans la facilité d'amélioration. Il est tout a fait possible que la société pour laquelle vous avez réalisé un soft, vous demande la version 2.00 plusieurs années après la version 1.00. Les infâmes bidouilles pour gagner 3 cycles d'horloges vous semblerons alors bien moins claires qu'un appel classique à la ligne A, peut être un peu plus lent. Mais là encore, il faut penser en se mettant à la place de l'utilisateur ! Ainsi SPECTRUM 512 est un excellent soft pensé avec un esprit programmeur et le résultat c'est que tout le monde utilise DEGAS...

Les routines ligne A sont donc des routines graphiques de base. Elles sont au nombre de 16, et sont numérotés de 0 à 15. Voici leur fonctions par numéro:

```
0   =   initialisation
1   =   placer un pixel à une couleur donnée
2   =   demander la couleur d'un pixel
3   =   tracé d'une ligne quelconque
4   =   tracé d'un ligne horizontale
5   =   tracé d'un rectangle rempli
6   =   remplissage d'une ligne d'un polygone
7   =   transfert d'un bloc de bit
8   =   transfert de matrice de caractère
9   =   visualisation de la souris
10  =   non-visualisation de la souris
11  =   transformation de la forme de la souris
12      =effacement de sprite
13  =   affichage de sprite
14  =   copie de zone (FDB)
15  =   remplissage de zone
```

Nous avons répondu à la première question: "qu'est ce que ça fait?" Nous pouvons passer à la suivante :"comment l'appelle-t-on ?" Un appel ligne A se fait à l'aide d'un word. Le poids faible de ce word contient le numéro de la fonction, et le poids fort est équivalent à $A (d'où le nom ligne A). Ainsi pour appeler la fonction 3, on utilisera le word $A003. Mais où le placer? Et bien tout simplement dans notre programme ! Pour pouvoir le noter tel quel nous le ferons précéder de DC.W de façon à ce que DEVPACK ne cherche pas à transformer ceci en code 68000 puisqu'aucune mnémonique ne correspond à ce nombre! Pour appeler la fonction 1 de la ligne A nous mettrons donc dans notre programme:

```
 DC.W $A001
```

Note: si cela vous intéresse, vous pouvez suivre les appels ligne A de la même manière que les appels Trap, sous MONST, en

tout cas si vous avez une des dernières versions de MONST.

Question suivante: comment passer des paramètres. En effet si nous voulons utiliser par exemple la fonction 2 pour connaître la couleur d'un pixel, il faudra bien fournir les coordonnées de celui-ci. Nous avons vu avec Gemdos, Bios et Xbios que les paramètres étaient passés par la pile. Nous avons également utilisé ce principe dans nos propres routines. Empilage, saut à la routine (par BSR ou TRAP) puis correction de la pile. Concernant la ligne A et le GEM, la pile n'est pas utilisée. En effet, la ligne A et le GEM utilisent un système de tableau. Nous avons étudié les tableaux et nous avons vu comment lire dans ceux-ci. Et bien c'est un peu ce principe qui va être utilisé. Les paramètres, parfois très nombreux, vont être placés dans un ou plusieurs tableaux, à des endroits précis, avant l'appel de la fonction choisie. Celle-ci ira chercher les paramètres nécessaires et retournera ensuite ses résultats également dans des tableaux. L'avantage du système des macros devient évident: si pour des appels gemdos, bios et xbios il suffit d'empiler (le nombre de paramètres n'est pas énorme et le principe et toujours le même), il faut en revanche une sacré dose de mémoire pour se rappeler dans quel tableau et surtout où dans ce tableau doivent être déposés les paramètres de chaque fonction. Malheureusement dans la bibliothèque de Devpack, il n'y a pas de macros pour la ligne A. Nous allons donc détailler quelques fonctions 'à la main'. Etant donné que vous avez sous la main le cours sur les macros, je ne peux que vous conseiller de réaliser les macros correspondantes à ces fonctions. Nous allons tout d'abord tracer un rectangle sur l'écran. Voir listing numéro 5.

Tout d'abord inclusion de la routine de démarrage des programmes, petit message de présentation en effaçant l'écran au passage. Ensuite initialisation de la ligne A. En retour, nous avons en A0 l'adresse du tableau que nous allons remplir avec les paramètres. Ceux-ci sont assez nombreux et doivent être passés à des endroits bien précis. Certains demandent une petite explication: Le clipping. C'est un système bien utile, que nous retrouverons dans le GEM. Par exemple nous voulons afficher sur l'écran une image, mais celle-ci ne doit apparaître que dans un petit rectangle. Il va donc falloir se compliquer sérieusement la tache afin e de ne pas afficher toute l'image. Au lieu de cela nous pouvons utiliser le clipping. Cela consistera à donner les coordonnées du rectangle dans lequel nous voulons que l'image apparaisse, et à dire que c'est le rectangle de clipping. Ensuite il reste à afficher l'image sur tout l'écran et elle n'apparaîtra que dans le rectangle, le reste de l'écran étant 'clippé'. La encore, il est bien évident que cela ralenti l'affichage, nous avons en effet affiché tout une image pour n'en voir qu'un bout, et de plus le système a été obligé de tester sans arrêt pour savoir s'il devait nous montrer ce pixel, puis l'autre etc... Cette méthode est pourtant hyper-utile et nous verrons cela de nombreuses fois.

Le type d'affichage. C'est un descripteur permettant de savoir comment va se faire l'affichage. 0= mode remplacement, 1=mode transparent, 2=mo deXOR, 3=transparent inverse. Essayer de faire 2e rectangle qui se recouvre et observez le résultat en faisant varier les modes.

Nous allons maintenant utiliser une autre fonction, qui demande la couleur d'un pixel. Dans le cas du traçage d'un rectangle nous avons utilisé la tableau dont l'adresse était en A0 pour passer les paramètres. Pour la demande de couleur d'un pixel et pour le coloriage d'un pixel, nous allons utiliser d'autres tableaux.Alors

là, suivez bien parce que les tableaux que nous allons décrire maintenant sont également utilisés pas le GEM? nous allons donc faire d'une pierre deux coups !!!!!

Tout comme le GEM donc, la ligne A utilise des tableaux, destinés chacun à recevoir ou à rendre des choses différentes. Le premier tableau, c'est le tableau CONTRL (Control) Il reçoit le numéro de la fonction et quelques autres paramètres. Le second c'est le tableau INT_IN. Cela signifie Integer In, c'est donc un tableau qui va recevoir des entiers (un nombre entier c'est en Anglais un integer) Le troisième tableau c'est PTSIN (Points In). C'est un tableau destiné à recevoir les coordonnées des points ou bien des dimensions. Disons que ce tableau va recevoir par exemple les coordonnées X et Y pour le centre d'un cercle, mais aussi son rayon. Il règne une certaine confusion dans les explications sur ce tableau. En effet il reçoit entre autre des coordonnées, qui vont souvent par couple (X et obligatoirement Y), ce qui fait que bien souvent on dit par exemple "il faut mettre 1 donnée dans PTSIN" alors qu'en fait il faut mettre un couple de données! Le quatrième tableau c'est INTOUT. C'est la même chose que INT_IN sauf que c'est en sortie, pour les résultats. Vous vous en doutez maintenant, le dernier tableau, c'est PTSOUT!!!!

Jetons maintenant un coup d'oeil sur le listing numéro 6.Un peu de réflexion et cela ne doit pas vous poser de problème. Essayez cependant de refaire ce listing avec des boucles parce qu'avec le coloriage d'un seul pixel, c'est bien sur un peu limité... Juste une chose, essayer de bien imaginer ce qui se passe avec les tableaux Contrl, Int_in, Ptsin, Intout et Ptsout parce dans le chapitre suivant il va y en avoir beaucoup.....

Bon maintenant que vous savez tracer un rectangle, vous pouvez également tracer des lignes, demander la couleur d'un pixel etc... La fonction la plus délicate est à mon avis celle qui sert pour l'affichage de texte. Elle permet cependant des effets assez intéressants (écriture avec différents styles).

Pour utiliser maintenant la ligne A, reportez vous aux descriptifs donnés dans la Bible et essayez! 2e remarque cependant: avec le GEM et la ligne A nous abordons en quelque sorte le concept du multi-utilisateur/multi-machines. En effet le GEM utilise beaucoup la ligne A et partage donc avec lui les tableaux. Or, votre programme peut très bien être en train de tracer de jolis rectangles lorsqu'il vient à l'utilisateur l'idée de promener sa souris dans le menu déroulant... Appel à GEM, modification des tableaux communs au GEM et à la ligne A, et au retour adieu les rectangles...

Là encore prudence et réflexion pour se mettre à la place de l'utilisateur...
Seconde remarque, les adresses fournies par la ligne A.

Nous pouvons prendre comme exemple les fontes. Où se trouvent-elles? Avec MONST, en se balladant en mémoire, il sera toujours possible de les trouver. Malheureusement elles ne sont pas au même endroit dans toutes les machines. Ce n'est pas grave, il suffit de passer par la ligne A. En effet la fonction $A000 permet d'initialiser mais en grande partie cela ne fait que nous fournir des adresses. Grâce à celles-ci nous pouvons en déduire de très nombreuses choses (emplacement des fontes entre autre). Là encore, il est préférable de passer par DC.W $A000 puis de faire des décalage pour trouver ce qui nous intéresse. Le petit rigolo qui se vante d'avoir trouvé l'adresse cachée qui donne les fontes se

rendra bien vite compte que cela ne marche pas tout le temps alors qu'avec la ligne A, c'est moins frime mais c'est sûr!

Pour l'utilisation de la ligne A, le descriptif des fonctions dans la bible est suffisant. Commencez par les rectangles, les lignes, placez des points où demander leur couleurs etc... Voici un exemple qui a été dans les premiers que j'ai réalisé avec la ligne A: Affichage d'un petit texte en haut à gauche (environ 5 lignes de 20 caractères). A l'aide d'un boucle, on demande la couleur des pixels, et on recolorie les pixels sur la droite de l'écran et de façon à ce que le texte apparaissent verticalement.

Pour ce qui est de déterminer l'emplacement de certaines données grâce à un saut dans les adresses d'initialisation ligne A, la bible est un peu silencieuse là-dessus alors que le Livre du Développeur chez Micro-App est plus fourni.

Conseil de dernière minute: faites beaucoup d'exercices avec la ligne A et surtout faites vous une bibliothèque de Macro pour ses fonctions.

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Seconde série                          *
*                                                                *
*                         Cours numéro 8                         *
******************************************************************
```

J'espère que vous n'abordez pas ce cours immédiatement après avoir fini de lire le précédent, mais, qu'au contraire, vous vous êtes plongé dans la ligne A avec bonheur!

Nous allons maintenant aborder un sujet qui, je dois l'avouer, m'effrayait un peu au début: le GEM.

En fait, en programmation comme dans beaucoup d'autres sujets, on se met à bosser simplement lorsqu'on y est obligé. Ainsi je n'avais réalisé que de tout petits trucs avec le GEM en assembleur, des trucs du genre ACCessoire de bureau qui permet de mettre l'écran en inverse vidéo, avec un choix dans une boîte d'alerte, avant de me trouver face au descriptif du soft que me proposait une boîte Parisienne. Je dois avouer que les premiers jours ont été un peu durs, mais qu'à la longue j'ai découvert que le GEM est vraiment une chose fabuleuse, pas très dur à maîtriser et que les possibilités sont largement suffisantes pour combler de bonheur tout programmeur digne de ce nom!!! J'espère donc que vous prendrez autant de plaisir que moi à programmer sous GEM une fois que vous aurez lu ce chapitre.

Tout d'abord les remarques qui se trouvent au début du chapitre

sur la ligne A, méritent d'être relus. Chercher donc de la documentation sans vous soucier du langage. Vous trouverez dans le dernier cours de cette série une bibliographie dans laquelle j'ai essayé de recenser divers articles bien sympas.

Le GEM est accessible par le TRAP #2, mais en fait grâce à ce trap nous avons accès à plusieurs parties du GEM, un peu comme si le trap #1 débouchait en même temps sur GEMDOS et BIOS. Le GEM est, en effet, composé principalement de 2 parties:

l'AES et le VDI.

Les routines du VDI - VIRTUAL DEVICE INTERFACE -
s'occupent des graphismes;

traçage de cercles,
ellipse,
traits,
rectangles
etc ...

Par contre, l'affichage des objets typiques du GEM :

fenêtres,
boîtes d'alertes

est réalisé par l'AES - APPLICATION ENVIRONMENT SYSTEM - .

Au fur et à mesure que vous découvrirez les fonctions VDI et AES vous vous rendrez compte de l'interaction des 2.

Le gros avantage de ce système peut s'expliquer simplement: il semble évident à tout le monde que sortir un dessin sur un écran, une disquette ou une imprimante, ce n'est pas la même chose.

ERREUR! Pour le GEM c'est pareil!

En fait, on peut dire que vous avez à préparer votre dessin et à dire "je l'envoie". Où? Ah là, c'est comme vous voulez parce que pour le VDI

l'écran,
la table traçante,
le disque,
l'imprimante
etc..

ne sont que des périphériques. Il suffit de lui désigner le périphérique de notre choix et il fait le boulot tout seul!!!!

Le seul problème:
les routines véritablement indépendantes des périphériques se trouvent dans GDOS - ce nom vous fait sûrement frémir d'effroi...-

-- Plusieurs programmes en même temps ?
Le principe du GEM est très attirant mais doit nous amener à réfléchir un petit peu sur le mélange potentiel des données. En effet le GEM, même s'il n'est pas réellement multitâche (être multitâche consiste à pouvoir faire tourner des applications exactement en même temps, ce qui est de toute façon impossible avec un 68000) le GEM donc, permet d'avoir plusieurs programmes actifs en même temps, en l'occurrence un programme principal et 6 accessoires. Pour le GEM il n'y a fondamentalement pas de différence entre un

PRG et un ACC et, pour cette raison, dans le vocabulaire GEM on les nomme "applications". Le GEM peut donc se retrouver avec 7 applications à gérer, chacune ayant potentiellement plusieurs fenêtres. On imagine facilement le bazar dans la mémoire!!!

-- Comment le GEM s'y retrouve t-il?

Au démarrage d'une application, celle-ci appelle la fonction APPL_INIT (Application_Init). Cette fonction prévient le GEM que notre application désire devenir active, et le GEM répond en nous fournissant un numéro, l'APP_ID (Application_Identificator) qui nous servira maintenant de référence. Notre application sera donc par exemple la 3, et sera reconnue comme telle par le GEM. Attention, l'APP_ID n'est donné à l'application que pour la durée de présence en mémoire de celle-ci. Si je lance un ACC le lundi, il peut hériter de l'APP_ID 4 mais si je le relance le mardi dans des conditions différentes, il peut obtenir l'APP_ID 5. Seulement il gardera cet APP_ID pendant toute son activité.

Lorsque l'on ouvre une fenêtre, c'est le même principe. Le GEM fournit ce que l'on nomme un HANDLE, c'est à dire un numéro correspondant à la fenêtre. En fait le HANDLE c'est le numéro d'identification d'une fenêtre mais potentiellement aussi d'un périphérique.

-- Multitraitement ?

Une des parties les plus importantes du GEM concerne la gestion des événements. Un événements c'est une action de l'utilisateur de la machine: clic souris, appui sur une touche etc... En fait, sous GEM les applications passent la plus grande partie de leur temps à attendre. Imaginons un PRG avec une fenêtre ouverte et un menu déroulant, et en même temps en mémoire 3 accessoires. Je vais dans le menu des accessoires. Le menu se déroule et les titres passent en inverse vidéo lorsque la souris passe dessus. Tout ceci est fait par le GEM, tout seul comme un grand. Je clique sur le titre de l'accessoire 2.

Le GEM détecte le clic, sait que je suis dans les accessoires, regarde sur quelle entrée j'ai cliqué. Comme c'est le GEM qui a attribué les entrées aux ACC, il envoie un message à l'accessoire 2 pour lui dire "il faut que tu t'ouvres". Celui-ci s'ouvre donc, et affiche une fenêtre à l'écran. Nous avons donc 2 fenêtres en même temps. Je clique sur celle du PRG, le GEM regarde à qui appartient cette fenêtre et envoi au PRG un message pour lui dire "on a activé ta fenêtre numéro XX" etc... Nous allons donc construire dans nos programmes sous GEM, un noyau de surveillance des actions, et nous réagirons en fonction de celles-ci.

-- Pour appeler les fonctions du GEM.

Les appels se font avec des paramètres que nous passons dans des tableaux. Comme nous utilisons DEVPACK, nous utiliserons sa bibliothèque. En effet, pour les appels au GEM, les paramètres sont nombreux et à placer à des endroits bien précis dans les tableaux. Pour ces raisons, les macros trouvent ici tout leur intérêt. Nous appellerons donc les fonctions du GEM avec les macros contenues dans la bibliothèque de DEVPACK. Cependant il convient de connaître également les tableaux, afin de pouvoir parfaitement comprendre le principe de fonctionnement.

Pour l'AES, il faut 6 tableaux:

CONTROL,
GLOBAL,
INT_IN,
INT_OUT,
ADDR_IN,
ADDR_OUT.

Pour le VDI il en faut 5:

CONTRL,
INTIN,
INTOUT,
PTSIN,
PTSOUT.

Attention à la légère différence d'orthographe entre INT_IN et INTIN ! Pour indiquer, lors de l'appel au GEM, l'emplacement où il trouvera ces tableaux, leurs adresses sont placées dans 2 listes:

Une pour l'AES:
aes_params dc.l control, global, int_in, int_out,addr_in, addr_out
et une pour le VDI:
vdi_params dc.l contrl, intin, ptsin, intout, ptsout

Voici les tableaux tels qu'ils sont définis dans la bibliothèque GEM de DEVPACK:

```
control   ds.w  5
global    ds.w  14
int_in    ds.w  16
int_out   ds.w  7
addr_in   ds.l  3
addr_out  ds.l  1

contrl    ds.w  1
contrl1   ds.w  1
contrl2   ds.w  1
contrl3   ds.w  1
contrl4   ds.w  1
contrl5   ds.w  1
contrl6   ds.w  1
contrl7   ds.w  1
contrl8   ds.w  1
contrl9   ds.w  1
contrl10  ds.w  1
contrl11  ds.w  1

intin     ds.w  128  min  30
intout    ds.w  128  min  45
ptsin     ds.w  128  min  30
ptsout    ds.w  128  min  12
```

Vous remarquez l'étrange mise en place du tableau CONTRL (tableau pour le VDI). En effet il faut assez souvent passer des paramètres 'à la main' dans ce tableau. Avec ce système, il sera possible de réaliser une opération du genre MOVE.W #10,CONTRL2

Attention cette numérotation des CONTRL correspond au nombre de words car chacune des entrées est définie par DS.W et non pas par DS.B!!! Ceci s'explique par le fait que 99% des documents relatifs au GEM sont pour le 'C' et que c'est ce type de commande qui est

faite en 'C'. Ainsi vous trouverez souvent; "mettre 10 en contrl(2)"; il vous suffira de faire MOVE.W #10,CONTRL2 et le tour est joué.

-- Pourquoi tant de tableaux? Parce que chacun de ces tableaux est destiné à recevoir ou à renvoyer un certain type de données. Voyons un descriptif rapide de ces tableaux.

Tableaux AES

** control

Destiné à contenir, dans l'ordre:

numéro	de la	fonction (.W),
nombre	d'octets de	INT_IN (.W),
nombre	d'octets de	INT_OUT (.W),
nombre	de longs mots de	ADDR_IN (.W)
nombre	de longs mots de	ADDR_OUT

** global
Ce tableau est un peu spécial, et les explications le concernant ne sont d'aucune utilité dans le cadre de ces cours. Une fois bien ingurgité le reste, vous pourrez toujours vous pencher dessus mais actuellement cela ne ferait que vous embrouiller! (Voir la bibliographie pour en savoir plus)

** int_in
Ce tableau est destiné à recevoir des valeurs entières (Integer In) dont l'AES aura besoin.

Par exemple un numéro de fenêtre.

** int_out
A l'inverse du précédent ce tableau renvoi des résultats par exemple si vous demander la surface disponible pour une fenêtre, vous aurez ici les dimensions.

** addr_in
Liste dans laquelle vous pourrez placer des adresses

** addr_out et là, vous pourrez en lire! En fait il suffit, comme toujours, de lire le nom du tableau pour en déduire ce qu'il contient!

Tableaux VDI

** contrl
Dans contrl0 nous plaçons l'opcode de la fonction, c'est ainsi que l'on appelle son numéro.

Dans contrl1 nous plaçons le nombre de données qui seront mise dans le tableau intin

Dans contrl2 nous plaçons le nombre de points (donc de couples de données) qui seront placés dans le tableau ptsin

Dans contrl3 nous placerons l'identificateur de la sous-fonction. Par exemple le traçage des cercles, ellipse, rectangle etc... ce fait avec la fonction 11. Il faut donc un sous-code permettant de définir quel module dans la fonction 11 on désire utiliser.

Dans contrl6 nous plaçons le handle du périphérique. (voir plus haut)
A partir de contrl7 il y a parfois des informations à passer, suivant la fonction.

** intin et ptsin
Nous remplirons ensuite le tableau INTIN avec les paramètres entiers demandés par la fonction et le tableau PTSIN avec les coordonnées demandées par la fonction. Une fois la fonction appelée, nous récupèrerons:

en contrl2 le nombre de couples de coordonnées de ptsout et
en contrl4 le nombre de mots de intout.

Voici un appel au GEM. Cette fonction affiche une phrase où l'on veut sur l'écran, contrairement à Gemdos (9) qui est limité sur les lignes et les colonnes de texte. De plus cette fonction permet d'afficher du texte avec des effets (gras, souligné etc...) avec la fonte de notre choix:

```
         LEA       MESSAGE,A1            adresse de la phrase
         LEA       INTIN,A0              tableau de réception
         MOVEQ.L   #0,D2                 init le compteur de lettres
ICI3     CLR.W     D0
         MOVE.B    (A1)+,D0              prélève en bytes
         BEQ       GO_ON                 fin du texte
         ADDQ.B    #1,D2                 on compte les lettres
         MOVE.W    D0,(A0)+              transfère dans INTIN (en
words)
         BRA       ICI
GO_ON    MOVE.W    #100,PTSIN            position X
         MOVE.W    #150,PTSIN+2          position Y
         MOVE.W    D2,CONTRL3            nombre de lettres
         MOVE.W    #1,CONTRL1
         MOVE.W    CURRENT_HANDLE,CONTRL6
         MOVE.W    #8,CONTRL             opcode
         MOVE.L    #VDI_PARAMS,D1
         MOVE.W    #$73,D0
         TRAP      #2
```

AAAAAAAAAARRRRRRGGGLLL!!!!!! C'est l'horreur n'est ce pas!

Il faut vous munir de patience et décortiquer les multiples appels au GEM.

Pour vous aider dans vos essais, vous trouverez ci-joint un dossier intitulé GEM. Il contient le source d'un ACC qui affiche une boîte d'alerte, d'un PRG qui gère une ressource (listing issu de WERCS) et d'un accessoire gérant une fenêtre. Pour ce qui est des éditeurs de ressources, je travaille avec K Ressource qui est bien sympa et qui ne plante pas, contrairement à Wercs!

Essayez de bien comprendre le principe: affichez une boîte d'alerte avec un seul bouton, puis avec plusieurs, faites des petites ressources toutes simples, puis de plus en plus compliquées etc... Le principe restera toujours le même quelle que soit la taille de votre application. Essayez de faire de petits acces-

soires du genre formateur de disquettes, avec un petit formulaire qui demande si l'on veut formater le disque A ou le B, en simple ou en double face etc...

A l'aide de ces exemples, vous devriez réussir à vous débrouiller. C'est avec ça que j'ai commencé! Jetez également un coup d'oeil sur la bibliographie, il y a quelques trucs intéressants!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*             COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                *
*                 par Le Féroce Lapin (from 44E)                 *
*                                                                *
*                         Seconde série                          *
*                                                                *
*                         Cours numéro 9                         *
******************************************************************
```

Ce petit cours va être un peu spécial, car il va fournir des indications sur la façon de réaliser des programmes GEM qui fonctionnent correctement.

Il y a en effet quelques "trucs" à respecter. Sur MAC, les programmeurs ont à leur disposition des bouquins traitant de l'ergonomie des logiciels. Qu'est ce que c'est ? Et bien c'est simplement un ensemble de règles à respecter afin que l'utilisateur ne soit pas perdu d'un programme à l'autre. Il faut en effet bien se souvenir que vous êtes programmeur et que votre ouvrage sera utilisé par des utilisateurs! Réfléchissons un peu: qu'est ce que l'utilisateur recherche dans un programme de dessin:

1) avoir une sauvegarde de fichier compressé avec un algorithme spécial, qui compresse plus que les copains.

2) avoir la possibilité de récupérer le plus facilement possible ces dessins dans d'autres softs.

Il paraît évident que c'est la seconde solution qui est la bonne! Pourtant nous assistons à une prolifération de formats de fichiers, tous plus débiles les uns que les autres! Ah bien sûr, le fichier compressé avec le logiciel de dessin Bidulmuche est plus petit que le fichier issu de Degas, mais il n'est reconnu par personne! Pourquoi chercher à épater la galerie avec des nouveaux formats ? Pourquoi ne pas charger et sauver en PI1, PI2, PI3, PC1, PC2, PC3 et c'est tout ? Première règle: il faut donc penser à l'utilisateur et ne pas chercher de trucs tordus! Le premier menu c'est celui du copyright, le second celui des fichiers avec tout en bas l'option Quitter. Quelle merde de devoir chercher l'option Quitter tout au bout d'un menu parce que le programmeur a voulu se distinguer!

De plus, par convention, les entrées de menus qui débouchent sur un dialogue seront suivis de 3 points.

Pensez aux grands écrans et au TT !!!!! Lorsque vous tapez dans des adresses système documentées, prévoyez un adressage sur 32 bits. Par exemple $FF8800 marchera sur ST mais plantera sur TT. C'est en effet une adresse invalide si on cherche à s'en servir en 32 bits (avec le 68030). Il faut donc utiliser $FFFF8800 qui marchera sur toutes les machines.

Ne testez pas la résolution avec Xbios4 ! C'est périlleux car, en cas de grand écran, vous recevrez n'importe quoi! Pour l'ouverture maxi d'un fenêtre, demandez au GEM la taille de la station de travail (voir le source avec la fenêtre). Une copie de blocs à faire ? Utilisez la fonction Vro_cpy, mais si c'est une copie avec l'écran, il y a une solution simple: vous serez obliger de fabriquer une structure FDB (Form Definition Block). C'est une structure qui indique la largeur de la surface de travail, sa longueur, le nombre de plans etc... Au lieu de demander au GEM toutes les données, remplissez la de 0, et le GEM saura de lui même que vous voulez parler de l'écran, et se débrouillera tout seul!

Pour vos accessoires, testez vos ressources en basse résolution! Un accessoire, comme son nom l'indique doit être accessoire, c'est-à-dire fonctionner sans gêner le reste de la machine : Petite taille (un accessoire de formatage de 100Ko hum!!!).

Un seul fichier ressource. Cela implique de ne pas utiliser de dessins et de construire sa ressource avec l'option SNAP active dans K_Ressource. Cette option permet d'avoir les boutons bien placés quelle que soit la résolution d'utilisation de la ressource. Si possible placez la ressource à l'intérieur de l'accessoire en la relogeant (voir listing de relocation ci-joint) cela évite d'avoir plusieurs fichiers à manier quand on déplace les accessoires.

N'hésitez pas à mettre des raccourcis clavier dans vos ressources. Si vous utilisez K Ressources vous verrez qu'il y a un accès à des bits inutilisés pour les objets. En effet, si l'on prend, par exemple, les bits définissant les flags de l'objet, on se rend compte que seul les bits 0 à 8 sont utilisés. Or le codage est fait sur un word, il nous reste donc 7 bits de libres. Ces bits sont laissés au programmeur pour y stocker ce qu'il veut. A titre indicatif voici ce que j'y mets:

```
Extended_type    scan code de la touche de raccourci pour
                 cet objet.

Extended_flag    Touche(s) devant être enfoncée simultanément pour
                 rendre ce raccourci valide.
                 Bit 15 -> Alternate
                 Bit 14 -> Control
                 Bit 13 -> Shift gauche
                 Bit 12 -> Shift droit
                 Bit 11 et 10 -> position du soulignement.
                 (0=pas de soulignement, 1=on souligne la première
                 lettre etc...)

Extended_state   Indication de la musique sur un octet
                 associé à la sélection de cet objet
                 0 pas de musique
                 1 clochette
                 2-63 digits
                 64-127 Xbios(32)
                 128-190 son au format Gist
```

191-255 musique format Mad Max ou Musexx

Tout ceci me permet d'avoir des raccourcis clavier inscrits DANS la ressource ce qui permet des modifications ultra-rapide. Pour les raccourcis, il faut utiliser de préférence la touche Alternate, car son utilisation avec un autre touche ne génère aucun caractère. Cependant, 6 raccourcis claviers utilisent Control. Ils sont issus du MAC et ont tendance à se généraliser. Ces raccourcis sont utilisés dans les formulaires avec champs éditables (entre autres choses) afin de faire du couper/coller entre ces champs.

```
CONTROL X        | pour couper (place en buffer en l'effaçant
                 | au préalable)

SHIFT CONTROL X | pour couper (mettre à la suite dans le buffer);

CONTROL C et     | Comme  avec  X sauf que,  dans le cas de X,  le
SHIFT CONTRL C | champ éditable est  effacé, alors qu'avec C, il
                 | conserve son contenu.

CONTROL V        | colle le contenu du buffer en effaçant au
                 | préalable le champ éditable.

SHIFT CONTROL V | idem mais sans effacement du champ éditable.
```

Une autre remarque concernant les formulaires avec plusieurs entrées éditables: j'ai remarqué que par habitude l'utilisateur tapait RETURN quand il avait fini de remplir un champ, et que, souvent, le bouton ANNULER est mis en défaut: l'appui sur RETURN donc, fait sortir du formulaire!

J'ai donc décidé de supprimer les boutons défaut lorsqu'il y a des entrées éditables et de gérer différemment RETURN qui permet alors de passer au champ éditable suivant (comme TAB).

J'ai également rajouté quelques autres touches. Alors qu'avec TAB, il est possible de passer d'un champ éditable au suivant, j'ai ajouté Shift+TAB pour remonter au champ éditable précédent. CLR HOME permettant de revenir au premier champ éditable du formulaire. Il serait possible d'ajouter UNDO.

Ré-écrire une gestion totale de formulaire (en prenant comme base de départ un article de ST Mag qui faisait ça en GFA par exemple) n'est pas très dur. Ce qui est sympa également c'est de rajouter un petit carré en haut à droite, afin de déplacer le formulaire.

Pour toutes ces options, vous pouvez bien sûr faire ça à votre propre sauce, mais les raccourcis clavier dont je parle sont déjà un peu utilisés. Autant continuer afin que le système se généralise, plutôt que de chercher à se distinguer par des trucs mal commodes.

Je terminerai ce chapitre sur le GEM en vous invitant à découvrir le Protocole de Communication GEM. Pour y avoir accès, décompactez le fichier PROTOCOL.AR avec ARCHIVE.PRG. Vous placez ce fichier en ram_disque (par exemple D). Vous préparez une disquette vierge, vous lancez Archive vous choisissez Unpack avec D:\PROTOCOL.AR et comme destination A:*.* et vous attendez.

Il y a tous les sources, la biblio, la doc, les exemples etc... Tous vos softs doivent être compatibles avec ce système s'ils veulent être dans le coup!!! C'est facile et cela permet des délires assez fabuleux!

Bons programmes GEM!!!!!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
*                                                                *
*                  par Le Féroce Lapin (from 44E)                *
*                                                                *
*                           Seconde série                        *
*                                                                *
*                          Cours numéro 10                       *
******************************************************************
```

Avant de vous fournir la bibliographie tant attendue, je vais vous parler très succinctement des interruptions. Le principe est normalement connu (sinon reportez vous au cours numéro 5 de la première série!). Il y a pourtant un sujet qui mérite un peu plus d'attention: le MFP68901. Il s'agit d'un circuit reconnu par le 68000 comme ayant un niveau d'interruption de 6 (voir feuille fournie avec le cours 1 et intitulée "TABLEAU DES VECTEURS D'INTER RUPTIONS DU 68000 ET DU 68901"). En interne ce circuit est capable de gérer 16 niveaux d'interruption, avec pour chaque niveau une adresse associée ($100-$13C pour le ST). Voici un petit bout de listing destiné à vous montrer comment placer sa propre routine dans le Timer A. Il faut d'abord bloquer les interruptions en plaçant le SR à $2700, sauver les registres du MFP, placer sa routine et ensuite redescendre le niveau d'interruption à son niveau habituel sur le ST, c'est à dire $2300. Ceci se fait bien évidemment en mode Superviseur, mode auquel on accède avec Gemdos $20.

```
* Pour mettre en place....
          MOVE.W    #$2700,SR             it interdites
          MOVE.B    $FFFFFA07,ANC_IERA    sauve les
          MOVE.B    $FFFFFA09,ANC_IERB    valeurs du
          MOVE.B    $FFFFFA13,ANC_IMRA    MFP 68901
          MOVE.B    $FFFFFA15,ANC_IMRB
          MOVE.B    $FFFFFA17,ANC_VR

          MOVE.L    $134,ANC_TIMEA        sauve Timer A
          MOVE.L    #MY_TIMEA,$134        place nouvelle routine

          CLR.B     $FFFFFA09             empêche interrupt 0-7
          CLR.B     $FFFFFA15             masque interrupt 0-7
          MOVE.B    #%00100000,$FFFA07    OK interrupt 13 (timer A)
          MOVE.B    #%00100000,$FFFA13    OK aussi pour son masque
          MOVE.B    #7,$FFFFFA19          règle TACR
          MOVE.B    #12,$FFFFFA1F         et TADR (1khz)
          BCLR      #3,$FFFFFA17          automatic end of interupt

          MOVE.W    #$2300,SR             autorise interruptions
* Et maintenant, lorsque nous quittons le programme...
```

```
          MOVE.W    #$2700,SR            it interdites
          MOVE.B    ANC_IERA,$FFFFFA07   restitution
          MOVE.B    ANC_IERB,$FFFFFA09
          MOVE.B    ANC_IMRA,$FFFFFA13
          MOVE.B    ANC_IMRB,$FFFFFA15
          MOVE.B    ANC_VR,$FFFFFA17
          MOVE.L    ANC_TIMEA,$134
          MOVE.W    #$2300,SR            autorise les interruptions

* Ma routine Timer A
* Ne pas oublier de sauver les registres qui sont utilisés
* et de terminer par un RTE.

MY_TIMEA: movem.l   d0-a6,-(sp)
'
'
'
          BCLR      #5,ISRA              suivant VR !!!
          MOVEM.L   sp)+,d0-a6
          RTE

*---------------------------------------------------------*
        SECTION BSS
ANC_TIMEA DS.L      1
ANC_IERA  DS.B      1             sauvegarde pour le MFP
ANC_IERB  DS.B      1                    "
ANC_IMRA  DS.B      1                    "
ANC_IMRB  DS.B      1                    "
ANC_VR    DS.B      1                    "
```

Je vais simplement vous décrire les registres IERA, IMRA, IPRA etc... du MFP. Pour de plus amples détails, consultez la Bible ST, le Livre du Développeur ou la doc officielle ATARI (l'idéal!), le but de ces cours n'étant pas d'être une recopie des infos que vous pouvez trouver autre part.

Tout d'abord les 16 niveaux du MFP sont classés en 2 groupes. Le A concerne les niveaux 15 à 8 et le B les niveaux 7 à 0. Attention, ici A et B n'ont rien à voir avec les Timers A et B!!! Pour chaque groupe (A ou B) on trouve une série de registres. Dans chacun de ces registres les interruptions sont représentées par un bit. Voyons ces registres:

IERA ($FFFFFA07) et IERB ($FFFFFA09)
Interrupt Enable Register A (ou B)
En plaçant à 0 le bit correspondant à une interruption, on interdit celle-ci. Elle ne sera aucunement prise en compte par le MFP.

IPRA ($FFFFFA0B) et IPRB ($FFFFFA0D)
Interrupt Pending Register A (ou B)
Lorsqu'une interruption arrive, le MFP met à 1 le bit correspondant à celle-ci dans IPRA (ou IPRB). Cela signale qu'une interruption est en attente. En effet à cause du système de priorité, il est possible qu'une interruption de haut niveau soit en cours et que pendant ce temps une interruption plus faible se déclenche. Il faut donc noter cette volonté de se déclencher, afin qu'une fois fini le traitement de l'interruption prioritaire, le traitement de la plus faible puisse s'effectuer.

Il est bien sûr possible de lire IPRA et IPRB afin de déterminer si une interruption est en attente. Il est aussi possible de met-

tre à 0 le bit d'une interruption en attente, afin que celle-ci ne se déclenche pas. Par contre, le fait de mettre un bit à 1 dans ce registre n'a aucun effet. On se rend ainsi compte que les possiblités deviennent déjà assez nombreuses. Une interruption peut ainsi en scruter une autre de façon à obtenir un cycle irrégulier d'interruption.

Il est en plus possible de laisser une interruption se déclencher juste 'pour voir'. ceci peut se faire en la laissant valide par IERA mais en la masquant avec IMRA.

IMRA ($FFFFFA13) et IMRB ($FFFFFA15)
Interrupt Mask Register A (et B)

Le masquage empêche une interruption de se déclencher bien qu'elle soit autorisée par IERA ou IERB. Ceci peut permettre par exemple à une interruption de niveau 4 de ne pas être gênée par une interruption de niveau 6. Pour cela, il lui suffit de masquer celle-ci durant son exécution.

VR ($FFFFFA17)
Vector Register

Cet octet est un peu spécial. Dans notre cas seul le bit 3 nous intéresse, les autres servant au MFP à coder le numéro de vecteur qui correspond à la source d'interruption. Le bit 3 sert à indiquer au MFP s'il est en mode Software End of Interrupt ou en mode Automatic End of Interrupt (mode par défaut). Voyons les explications avec les registres suivants:

ISRA ($FFFFFA0F) et ISRB ($FFFFFA11)
Interrupt in Service Register A (ou B)

Un bit à 1 indique que l'interruption est en cours de traitement. Si nous sommes en mode Software End of Interrupt, c'est à la fin de notre routine en interruption que nous devons indiquer, nous même que notre routine est finie. Pour cela il faut mettre à 0 le bit correspondant à notre interruption, dans ISRA (ou ISRB). Tant que ce bit est à 1, les interruptions moins prioritaires ne peuvent pas venir nous déranger. Par contre, dès que le traitement de notre interruption commence, son bit IPRA est remis automatiquement à 0, et pendant le traitement de cette interruption, une autre de même niveau peut très bien intervenir. Le bit d'IPRA sera donc remis à 1, mais cette nouvelle interruption ne sera traitée que lorsque la première aura remise le bit ISRA à 0.

D'un autre côté, si nous sommes en mode Automatic End of Interrupt, dès que notre routine s'exécute, le MFP met, bien sûr, son bit d'IPRA à 0 (puisque l'interrupt n'est plus en attente), mais met également son bit ISRA à 0 ! Il est alors possible que d'autres interruptions plus faibles viennent interrompre la première, même si elle n'est pas terminée.

Vous voyez qu'avec tout ceci, il est possible de jongler allègrement en envisageant les cas les plus tordus!!! Concernant les timers A, B, C et D, voici juste les adresses permettant de les adresser, les informations fournies dans la bible ou le livre du développeur étant largement suffisante. Si par hasard vous étiez avides d'informations sur ce circuit, précipitez vous chez un libraire spécialisé dans la "littérature" électronique et demandez

un ouvrage sur le MK68901. Passez aussi chez le pharmacien prendre quelques tubes d'aspirine....

Pour terminer ces cours d'assembleur, voici une petite bibliographie afin que vous puissiez diriger vos recherches vers les sujets qui vous intéressent.

Le Langage Machine sur ST (Ed. Micro App). Assez inutile comme bouquin! Très très peu d'informations, non vraiment, ce n'est pas un bon achat.

La bible ST (Ed. Micro-App). Devenue assez introuvable, dommage! Pas mal de bonne infos, bien suffisant dans la plupart des cas. A été remplacé par le Livre du Développeur.

Trucs et Astuces (Ed. Micro-App). A éviter absolument! Les programmes réalisés en suivant ces conseils seront sûrement incompatibles dès qu'il y aura changement de TOS.

Le Livre du Développeur (Tome 1 et 2) (Ed. Micro App) Si vous n'avez pas accès à la doc. pour les développeurs ayant l'agrément Atari, c'est le bouquin qu'il vous faut. On peut regretter les dizaines de pages contenant le listing du BIOS des STF et les autres dizaines de pages contenant le BIOS du Méga ST. Cela fait un peu remplissage: soit on a le niveau en assembleur pour y comprendre quelque chose et alors on a le niveau pour suivre le BIOS avec MONST, soit, de toute façon, on n'y comprend rien et ce n'est pas parce qu'on a le listing sous les yeux que cela va changer quelque chose. Enfin, c'est mon avis, et comme, en plus, le listing n'est valable que pour les modèles cités, si vous vous y fiez pour "découvrir" de nouvelles adresses vous risquez fort d'être surpris avec les machines plus récentes ...

Le livre du GEM sur Atari ST (Ed. Micro App). Si vous pensez apprendre à programmer GEM avec ça, vous courrez à la catastrophe. Le problème du GEM c'est qu'il est utilisable en assembleur à condition de se servir des macros, sinon c'est beaucoup trop lourd. Or dans un livre commercialisé, il est impossible de se borner à un seul assembleur. Ceci fait que les auteurs ont tout décortiqué avec les appels directs au TRAP #2, et que cela devient totalement incompréhensible pour le débutant. Par contre, les fonctions y sont relativement bien détaillées, et, si vous savez déjà ce que vous cherchez, ce livre sera un bon outil. On peut, simplement, regretter quelques absence comme les fonctions Escape du GEM et quelques fonctions de l'AES. A noter que ce livre est repris à 99% dans le Livre du Développeur.

Doc GFA 3.00. Là, c'est le bon truc! En effet dans les dernières pages de la doc de ce BASIC, vous trouverez la liste de toutes les fonctions GEMDOS, BIOS et XBIOS (il manque la fonction GEMDOS $20 permettant de passer en Superviseur, étant donné qu'elle est inaccessible en GFA), les codes ASCII, les codes clavier et quelques pages avant, une bonne liste des fonctions AES. Je dois avouer que j'ai photocopié ces quelques pages et qu'elles me servent bien souvent: les informations sont suffisantes dans le plupart des cas, et surtout très faciles à trouver, ce qui n'est pas le cas par exemple du Livre du GEM qui ne comprend même pas d'index des fonctions !!!

A noter que certaines fonctions du GEM existent dans la bibliothèque interne du GFA mais ne sont pas disponibles dans celle de DEVPACK. C'est le cas de Form_Button et Form_Keybd. Et c'est gênant! En effet au début Form_do n'existait pas. Les gens de Digital Re-

search ont donc décidé de fabriquer de toutes pièces cette fonction à partir de Form_Keybd et Fomr_button, et ont diffusé le source en C. Au bout d'un moment, les programmeurs n'ont plus utilisé que Form_Do et nous sommes tombés dans la routine de gestion de formulaires que nous connaissons actuellement.

Si vous voulez faire votre propre Form_do, plus évolué il "suffit" de reprendre Form_Keybd et Form_button pour recréer Form_do. Malheureusement ces deux fonctions sont tombées dans les oubliettes et pour les avoir pffuuuuuttt!!!

Vous voulez connaître leur opcode et les paramètres ? Facile comme tout! Si vous avez bien suivi ces cours, vous savez plein de chose sur le GEM (où se place l'opcode d'une fonction, où est inscrit le nombre de paramètres etc... et bien vous lancez AMonst, puis vous faites un petit bout de GFA qui appelle Form_keybd en affichant au préalable sur l'écran l'adresse du tableau AES, et qui attend ensuite un appui sur une touche. Tout ceci se passe sous l'interpréteur du GFA bien sûr! Dès que vous êtes sur l'attente d'un appui touche, vous déclenchez AMONST (shift+alternate+help) et hop, la grande recherche commence. Vous faites de même avec Form_button, et le tour est joué. Courage, ce n'est pas bien dur, et cela fait un excellent exercice!

Mise en oeuvre du 68000 (Ed. Sybex) Excellent bouquin, traitant du 68000 en général. Ne vous attendez pas à y trouver des adresses système pour votre ST. On parle, ici, cycles d'horloges, modes d'adressage etc...

Implantation des fonctions usuelles en 68000 (Ed. Masson, par François BRET). Comment faire des sinus, des cosinus, des transformées de Fourier etc.. . N'est pas vendu avec l'aspirine ni le café, mais s'avère indispensable, si vous vous attaquez aux problèmes de trigo en ASM. Seulement 191Frs, une misère! (Merci à Shizuka de m'avoir fourni la référence de cet ouvrage!!!!!)

STATION INFORMATIQUE 2 rue Piémontési 75018 PARIS
tél:(1)42.55.14.26
Excellent catalogue de Dom-Pubs. Avant de chercher à pirater des softs divers sans en avoir la doc, jetez un coup d'oeil sur leur catalogue. De nombreuses petites choses sont à y découvrir (sources divers en ASM, utilitaires de débuggage etc...) Si vous êtes un adepte de l'émulation Mac, dans les dom-pub de cette machine se trouve le pack Cyclan, un assembleur Mac avec son éditeur etc... Bien sympa pour découvrir cette autre machine, équipée elle aussi d'un 68000.

Les ST Mags fourmillaient également de trucs divers, en tout cas surtout dans les anciens numéros à mon avis...

Voici une petite liste non-limitative...

Intégration d'une ressource en GFA (46)
Form Exdo (46)
Echantillons sur ST (27,28,29,30,31,35) (En GFA avec de l'assembleur, mais le GFA on ne le sent qu'un tout petit peu...)

Scrolling en GFA (31 à 45 sauf numéro 37 et 44) Même remarque que pour les digits!!!

Gestion des disquettes en GFA (13,14,15,16)

Programmer GEM (6 à 30 environ) Super! Tout en C mais les appels

sont les mêmes et la syntaxe identique à celle de la biblio DEV-PACK! Faites vite une recherche de vos vieux numéros de ST Mag car pour apprendre le GEM c'est extra!!!

Animation en 3D (45,46,49) Pour le moment le dernier article n'est pas sorti... Espérons que cela ne fera pas comme le listing permettant de booter sur une cartouche....Bien clair, sympa, idéal pour commencer la 3D.

Pour les coprocesseurs, 2 articles dans les numéros 31 et 32.

Dans le défunt First (1ST) il y avait quelques trucs sympa sur le MIDI, le décompactage des images DEGAS, la création de ram-disque et de spooler d'imprimante.

Atari Mag revient en force en ce moment.

Très bon articles sur la programmation graphique du STE. Cela permet de se faire une idée sur le BLITTER, les 4096 couleurs etc... C'est en BASIC mais la traduction est facile. (21,22).

Etant dans la confidence, je peux vous informer également que dans les prochains numéros d'Atari Mag vous trouverez des articles sur différents sujets avec sources en ASM et en GFA et que ce type d'article devrait durer pas mal de temps.

Bon, en résumé, achetez tous les mois ST Mag et Atari Mag. Si besoin est, achetez les à plusieurs, mais faire un gros tas de toutes ces revues car c'est une mine d'information importante. Bien souvent les questions qui sont posées sur RTEL par exemple trouvent leur réponse dans une lecture des vieux numéros! Quelques bons classeurs, des intercalaires et vous aurez une documentation énorme pour pas très cher.

Je terminerai ces cours en renouvelant l'avertissement concernant la chasse aux listings! Il est préférable d'avoir quelques bouquins et des notes sur papier plutôt que 10 mégas de sources auquel on ne comprend pas grand chose, et qui de, toute façon, sont assez malcommodes à consulter par rapport à des informations écrites!

Pour ce qui est des programmes assez gros, préparez quelques feuilles à côté de vous, pour y noter les noms des routines, des labels, des variables etc... vous vous rendrez vite compte que la principale difficulté de l'assembleur réside dans le fait que les listings sont très très longs et qu'il est difficile de s'y déplacer pour chercher quelque chose. Programmez proprement, soyez clairs et évitez si possible les bidouilles. Commentez abondamment vos sources car dans 6 mois lorsqu'il faudra faire une petite modification, vous verrez la différence entre les sources clairs et les autres!

Allez, je vous quitte en espérant que ces cours vous auront intéressés et qu'ils vous auront donné goût à l'assembleur! N'oubliez pas que vous pouvez toujours me contacter sur le 3614 RTEL1 (ou RTEL2) en bal FEROCE LAPIN, et qu'il y a sur ce serveur une rubrique pour l'assembleur 68000 sur ST, MAC ou AMIGA. Pour accéder à cette rubrique, tapez *MOT et ENVOI. A bientôt!

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
*                                                                *
*                  par Le Féroce Lapin (from 44E)                *
*                                                                *
*                          Seconde série                         *
*                                                                *
*                   Listing  numéro 1/ Cours 3                   *
*                                                                *
******************************************************************

* Affichage d'une image incluse dans le programme.

         LEA       IMAGE,A6           adresse image
         ADDA.L    #2,A6              saute l'en-tête DEGAS
         MOVE.L    A6,-(SP)           on est donc sur les couleurs
         MOVE.W    #6,-(SP)           mise en place par XBIOS(6)
         TRAP      #14
         ADDQ.L    #6,SP

         MOVE.W    #3,-(SP)           cherche l'adresse de l'écran
         TRAP      #14
         ADDQ.L    #2,SP
         MOVE.L    D0,A5              la sauve en A5

         ADDA.L    #32,A6             saute les couleurs (16 words)
         MOVE.W    #7999,D0           init le compteur
COPIE    MOVE.L    (A6)+,(A5)+        transfère image vers l'écran
         DBF       D0,COPIE

         MOVE.W    #7,-(SP)           attend un appui sur une touche
         TRAP      #1
         ADDQ.L    #2,SP

         MOVE.W    #0,-(SP)           et bye bye
         TRAP      #1

IMAGE    INCBIN    "A:\TRUC.PI1"      l'image
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
******************************************************************
*                                                                *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST             *
*                                                                *
*                  par Le Féroce Lapin (from 44E)                *
*                                                                *
*                          Seconde série                         *
*                                                                *
*                    Listing numéro 2/ Cours 5                   *
*                                                                *
******************************************************************
```

```
         INCLUDE    "D:\START.S"

         MOVE.L     #MESSAGE,-(SP)     message de présentation
         MOVE.W     #9,-(SP)
         TRAP       #1
         ADDQ.L     #6,SP

TOUCHE   MOVE.W     #7,-(SP)           attente appui sur une touche
         TRAP       #1
         ADDQ.L     #2,SP
         SWAP       D0                 pour avoir le scan code

         CMP.W      #1,D0              escape ?
         BEQ        FIN                oui donc bye bye
         CMP.W      #$3B,D0            par rapport à F1
         BCS        TOUCHE             en dessous donc pas valable
         CMP.W      #$44,D0            par rapport à F10
         BHI        TOUCHE             en dessus donc pas valable

* La touche est valable
         ADD.W      #6,D0              pour avoir asci de A,B,C...
         MOVE.W     D0,-(SP)           affiche
         MOVE.W     #2,-(SP)
         TRAP       #1
         ADDQ.L     #4,SP
         BRA        TOUCHE             et on recommence

FIN      CLR.W      -(SP)
         TRAP       #1
*----------------------------------------------------*
         SECTION DATA
MESSAGE  DC.B       27,"E","TAPEZ SUR UNE TOUCHE DE FONCTION",13,10
         DC.B       "ESCAPE POUR SORTIR",13,10,0
         SECTION BSS
         DS.L       256
PILE     DS.L       1
         END
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
*********************************************************************
*                                                                   *
*                COURS D'ASSEMBLEUR 68000 SUR ATARI ST              *
*                                                                   *
*                   par Le Féroce Lapin (from 44E)                  *
*                                                                   *
*                             Seconde série                         *
*                                                                   *
*                        Listing numéro 3/ Cours 3                  *
*                                                                   *
*********************************************************************
          INCLUDE    "B:\START.S"

          MOVE.L     #MESSAGE,-(SP)       message de présentation
          MOVE.W     #9,-(SP)
          TRAP       #1
```

```
          ADDQ.L     #6,SP

TOUCHE    MOVE.W     #7,-(SP)   attente appui sur une touche
          TRAP       #1
          ADDQ.L     #2,SP
          SWAP       D0         pour avoir le scan code

          CMP.W      #1,D0      escape ?
          BEQ        FIN        oui donc bye bye
          CMP.W      #$3B,D0    par rapport à F1
          BLO        TOUCHE     en dessous donc pas valable
          CMP.W      #$44,D0    par rapport à F10
          BHI        TOUCHE     en dessus donc pas valable

* La touche est valable
          LEA        TABLEAU,A0
          SUB.W      #$3B,D0    pour avoir un chiffre de 0 à 9
*         MULU.W     #2,D0      ATTENTION! pourquoi l'étoile début ??
          MOVE.W     0(A0,D0.W),D1
          MOVE.W     D1,-(SP)
          MOVE.W     #2,-(SP)
          TRAP       #1
          ADDQ.L     #6,SP
          BRA        TOUCHE

FIN       CLR.W      -(SP)
          TRAP       #1
*---------------------------------------------------*
          SECTION DATA
MESSAGE   DC.B       27,"E","TAPEZ SUR UNE TOUCHE DE FONCTION",13,10
          DC.B       "ESCAPE POUR SORTIR",13,10,0
          EVEN
TABLEAU   DC.W       "A","Z","E","R","T","Y","U","I","O","P"
          SECTION BSS
          DS.L       256
PILE      DS.L       1
     END
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
**********************************************************************
*                                                                    *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST                 *
*                                                                    *
*                  par Le Féroce Lapin (from 44E)                    *
*                                                                    *
*                          Seconde série                             *
*                                                                    *
*                    Listing  numéro 4/ Cours 6                      *
**********************************************************************
* Automodification de programme
         INCLUDE    "B:\START.S"      routine de démarrage
         MOVE.L     #MESSAGE1,-(SP)   on présente
         MOVE.W     #9,-(SP)          le programme
         TRAP       #1
         ADDQ.L     #6,SP
```

```
DEBUT    MOVE.W   #7,-(SP)          attend un appui touche
         TRAP     #1
         ADDQ.L   #2,SP
         CMP.W    #" ",D0           espace ?
         BEQ      FIN               oui donc bye bye
         MOVE.W   #$4E71,VARIANTE   place un NOP (par défaut)
         CMP.W    #"A",D0           appui sur 'A'?
         BNE      ICI               non
         BSR      ROUTINE           oui donc sub-routine longue
         BRA      DEBUT             et on recommence
ICI      MOVE.W   #$4E75,VARIANTE   place un RTS à la place
         BSR      ROUTINE           du NOP et va à sub-routine
         BRA      DEBUT             puis recommence

FIN      MOVE.W   #0,-(SP)
         TRAP     #1
*--------------------------------------------*
ROUTINE  MOVE.L   #MESSAGE2,-(SP)
         MOVE.W   #9,-(SP)
         TRAP     #1
         ADDQ.L   #6,SP
VARIANTE NOP
         MOVE.L   #MESSAGE3,-(SP)
         MOVE.W   #9,-(SP)
         TRAP     #1
         ADDQ.L   #6,SP
         RTS
*--------------------------------------------*
         SECTION DATA
MESSAGE1 DC.B     27,"E","A pour un grand message,",13,10
         DC.B     "espace pour sortir, autre touche",13,10
         DC.B     "pour un message plus court",13,10,0
         EVEN
MESSAGE2 DC.B     13,10,"Voila le petit message...",0
         EVEN
MESSAGE3 DC.B     "Oups, non, c'est le long!!!",0
         EVEN
         SECTION BSS
         DS.L     100
PILE     DS.L     1
         END
```

COURS D'ASM 68000

(par le Féroce Lapin)

retour au VOLUME 2

```
**********************************************************************
*                                                                    *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST                 *
*                                                                    *
*                  par Le Féroce Lapin (from 44E)                    *
*                                                                    *
*                           Seconde série                            *
*                                                                    *
*                     Listing  numéro 5 / Cours 7                    *
**********************************************************************

         INCLUDE  "B:\START.S"
```

```
        MOVE.L    #MESSAGE,-(SP)     coucou et efface écran
        MOVE.W    #9,-(SP)
        TRAP      #1
        ADDQ.L    #6,SP

        DC.W      $A000              initialisation ligne A

        MOVE.W    #1,24(A0)          couleur du premier plan
        CLR.W     26(A0)             couleur du second
        MOVE.W    #2,36(A0)          type d'affichage
        MOVE.W    #50,38(A0)         X1
        MOVE.W    #50,40(A0)         Y1
        MOVE.W    #200,42(A0)        X2
        MOVE.W    #300,44(A0)        Y2
        MOVE.L    #MOTIF,46(A0)      adresse du motif
        MOVE.W    #3,50(A0)          nombre de mots du motif (-1)
        CLR.W     54(A0)             pas de clipping
        DC.W      $A005              appel à la fonction

        MOVE.W    #7,-(SP)           attente appui touche
        TRAP      #1
        ADDQ.L    #4,SP
        MOVE.W    #0,-(SP)           et bye bye
        TRAP      #1
*----------------------------------------*
        SECTION DATA
MESSAGE DC.B      27,"E","Tracage de rectangle avec ligne A",0
        EVEN
* Motif, en .W Je l'ai écrit en binaire pour mieux voir le dessin.*
MOTIF   DC.W      %1100110011001100
        DC.W      %1100110011001100
        DC.W      %0011001100110011
        DC.W      %0011001100110011

        SECTION BSS
        DS.L      100
PILE    DS.L      1
        END
```

```
*********************************************************************
*                                                                   *
*              COURS D'ASSEMBLEUR 68000 SUR ATARI ST                *
*                                                                   *
*                  par Le Féroce Lapin (from 44E)                   *
*                                                                   *
*                          Seconde série                            *
*                                                                   *
*                     Listing  numéro 6 / Cours 7                   *
*********************************************************************

        INCLUDE   "B:\START.S"

        MOVE.L    #MESSAGE,-(SP)     coucou et efface écran
        MOVE.W    #9,-(SP)
        TRAP      #1
        ADDQ.L    #6,SP

        DC.W      $A000              initialisation ligne A
```

```
* A l'aide de A0 (retour de la fonction $A000 nous obtenons
* l'adresse des différents tableaux
* 4(A0) donne l'adresse du tableau Contrl
* 8(A0) donne l'adresse du tableau Int_in
* 12(A0) donne l'adresse du tableau Ptsin
* 16(A0) donne l'adresse du tableau Intout
* 20(A0) donne l'adresse du tableau Ptsout

         MOVE.L    8(A0),A3          sauve adresse de Int_in
         MOVE.L    12(A0),A4         sauve adresse de Ptsin

* On passe les coordonnées du point dont on veut la couleur
* On passe bien sur ces coordonnées dans Ptsin.

         MOVE.W    #100,(A4)
         MOVE.W    #122,2(A4)
         DC.W      $A002             demande couleur d'un pixel

* La fonction $A002 renvoie la couleur du pixel en D0, on recolorie
* un autre pixel avec la couleur du premier

         MOVE.W    #100,(A4)
         MOVE.W    #123,(A4)
         MOVE.W    D0,(A3)           couleur bien sûr dans INT_IN
         DC.W      $A001             coloriage d'un pixel

         MOVE.W    #7,-(SP)          attente appui touche
         TRAP      #1
         ADDQ.L    #4,SP
         MOVE.W    #0,-(SP)          et bye bye
         TRAP      #1
*----------------------------------------*
         SECTION DATA
MESSAGE  DC.B      "Coloriage de pixel avec ligne A",0

         SECTION BSS
         DS.L      100
PILE     DS.L      1
         END
```